Analysis of Algorithms II

BLG 336E

Project 2 Report

EREN CULHACI
150220763
culhaci22@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 04.04.2024

0.1. Implementation Details

```
distance(p1, p2):
    x_diff = p2.x - p1.x // Calculate the difference between x-coordinates of two points
    y_diff = p2.y - p1.y // Calculate the difference between y-coordinates of two points
    distance_squared = x_diff * x_diff + y_diff * y_diff // Calculate the squared distance
    distance = sqrt(distance_squared) // Calculate the square root to get the Euclidean distance
    return distance // Return the Euclidean distance


compareX(p1, p2):
    return p1.x < p2.x // Return true if the x-coordinate of the first point is less than the x-coordinate of the second point

compareY(p1, p2):
    return p1.y < p2.y // Return true if the y-coordinate of the first point is less than the y-coordinate of the second point
```

Figure 0.1: Distance and Compare Functions

- In the distance function, I first calculate the x coordinate difference by subtracting the x coordinates of two points, then I do the same thing to y coordinate of two points. Lastly I square both x difference and y difference then take the square root of their sum. After that i return the result distance.
- In compareX and compareY functions, I compare two points, If the first point's x or y coordinate is smaller than second point's x or y coordinate, I return bool value true. I used this for sorting purposes, I compared two points.
- Since i do not have any loop or recursion in all of these functions, and I have constant number of instructions, time complexity is $O(1)$ for all of them.

```
bruteForceClosestPair(points):
    closestPair = empty pair // Initialize a pair for the closest points
    minDist = maximum double value // Initialize the minimum distance to the maximum value of double

    for i = 0 to points.size() - 1: // Iterate over the points
        for j = i + 1 to points.size() - 1: // Iterate over the points
            dist = distance(points[i], points[j]) // Calculate the distance between the points
            if dist < minDist: // If the distance is less than the minimum distance
                minDist = dist // Update the minimum distance
                closestPair = pair(points[i], points[j]) // Update the closest pair

    return closestPair // Return the closest pair of points
```

Figure 0.2: Brute Force Method

- This brute force algorithm first creates an empty closestPair pair and minDist as minimum distance to be a big value at first.
- Then it iterates over the points and calculates the distance among them one by one. After a calculation, if the minimum distance is smaller than the current minimum distance, it updates the minDist with this disitance and updates the closestpair. After all iterations, the closestpair becomes the points with the closest distance in the points vector. Then I return that closestpair.
- Since this method iterates over all points and I have a nested for loop, its time complexity is $O(n^2)$. Also, the space complexity of the algorithm is $O(1)$. It only uses a constant amount of space for whatever the input is.

```
closestPair(points):
    if size of points is 3 or less:
        return bruteForceClosestPair(points)

    sort points by x-coordinate

    // Divide the points into two halves
    // O(1) time complexity
    mid = size of points / 2
    leftHalf = points[0 : mid]
    rightHalf = points[mid : end]

    // Recursively find the closest pair in each half
    // T(n) = 2T(n/2) time complexity
    leftClosest = closestPair(leftHalf)
    rightClosest = closestPair(rightHalf)

    // Find the minimum distance between the closest pairs in left and right halves
    // O(1) time complexity
    minDist = minimum of (distance(leftClosest.first, leftClosest.second),
                          distance(rightClosest.first, rightClosest.second))

    // Merge the results from left and right halves
    // O(n) time complexity
    if minDist is 0:
        // If the minimum distance is 0, return one of the pairs with 0 distance
        if distance(leftClosest.first, leftClosest.second) is 0:
            return leftClosest
        else:
            return rightClosest

    // Create a strip vector to store points within the minimum distance strip
    // O(n) time complexity
    strip = empty vector
    for each point p in points:
        if absolute value of (p.x - points[mid].x) is less than minDist:
            append p to strip

    // Sort points in the strip by y-coordinate
    // O(n log n) time complexity
    sort strip by y-coordinate

    // Compare points in the strip to find closest pair within the strip
    // O(n) time complexity
    for i from 0 to size of strip - 1:
        for j from i + 1 to size of strip - 1 and strip[j].y - strip[i].y is less than minDist:
            dist = distance(strip[i], strip[j])
            if dist is less than minDist:
                minDist = dist
                mergedClosest = (strip[i], strip[j])

    // Return the closest pair among left, right, and merged closest pairs
    if minDist is distance(leftClosest.first, leftClosest.second):
        return leftClosest
    else if minDist is distance(rightClosest.first, rightClosest.second):
        return rightClosest
    else:
        return mergedClosest

// T(n) = 2T(n/2) + O(n) overall.
// Using master theorem, closestPair function is overall O(n log n)
```

Figure 0.3: Closest Pair Function

- We have different steps in closestPair function.

1. **Divide Step**
   The points are divided into two halves at each recursive call. This step has a time complexity O(1) since it only has arithmetic operations.

2. **Recursive Calls**
   The function makes two recursive calls, one for the left part of points and one for the right half of them. Each recursive call operates on approximately half the number of points. So, we can tell that the time complexity is: $T(n) = 2T(n/2)$
3. **Merge (Conquer)**
   After the recursive calls, the function merges the results from the left and right halves. Time complexity of merge is $O(n)$, where n is the total points. It is because it iterates through strip vector, which may contain at most n points.
4. **Strips**
   There is a loop processes through the points in strips. Since this loop iterates over at most n points, and does constant time operations, totally, time complexity of this step is $O(n)$
5. **Total Overall Complexity**
   In total, if we combine these, we have the time complexity relation as this: $T(n) = 2T(n/2) + O(n)$
   And using master theorem we can conclude that this function's time complexity is $O(n * log(n)$

This function recursively divides the set of points into smaller subsets until there are only a few points left (generally 3 or fewer). For these small set of points, it uses a brute-force method to find the closest pair. Then it sorts the points by their x-coordinate and recursively finds the closest pairs in the left and right halves of the sorted points since we divided these points into halves in previous step. After that, it merges the results from the left and right halves and builds a strip of points around the mid line and finds the closest pair in this strip. Then lastly, it compares the distances of closest pairs from the left, right, and the merged strip, and returns the closest pair.

```
removePairFromVector(point_vector, point_pair):
    for each point in point_vector:
        if point matches either point in point_pair:
            remove point from point_vector

    return point_vector
```

Figure 0.4: Removing the found closest pair from the vector

- This function basically finds the "to be removed points" in the point_vector. Then it removes these points.

- Time complexity of this function is $O(n)$ since we iterate over the vector, where n is the size of the vector.

```
findClosestPairOrder(points):
    pairs = empty vector of pairs of Points
    unconnected = Point with coordinates (-1, -1)

    while size of points is greater than 1 do
        closestpair = closestPair(points) // Find the closest pair of points
        add closestpair to pairs // Add the closest pair to the pairs vector
        points = removePairFromVector(points, closestpair) // Remove the closest pair from the points vector

    if size of points is equal to 1 then
        unconnected = the remaining point in points

    for each pair in pairs do
        sort the points according to the point with the smaller y-coordinate, if equal smaller x coordinate.

    for i from 0 to size of pairs - 1 do
        print "Pair " + (i+1) + ": " + pairs[i].first.x + ", " + pairs[i].first.y + " - " + pairs[i].second.x + ", " + pairs[i].second.y

    if unconnected.x is not equal to -1 then
        print "Unconnected " + unconnected.x + ", " + unconnected.y
```

Figure 0.5: Wrapper function

- This function finds the closest pair in points vector by calling the closestPair function then adds this pair to pairs vector. Then calls the removePair function to remove the found pair from points.
- Then it determines the unconnected point by remaining points.
- Then it sorts the last pairs vector by the coordinates of pairs in this lastly created vector. So it ensures the point that has smallest y is the first element in the pair, if the y values are the same, it ensures the smaller x is the first.
- Then it prints the results.
- This is a wrapper function. So the complexities of the functions used in this function are explained also above separately.

```
readCoordinatesFromFile(filename):
    points = empty vector of Points

    open filename as file
    if file is not open THEN
        return points

    while there are lines left to read from file do
        line = READ line from file
        point = create a new Point
        parse coordinates from line into point.x and point.y
        add point to points vector

    CLOSE file
    return points
```

Figure 0.6: Reading from file

- This function basically first opens the file, checks if the file is properly opened and then it reads the file line by line. It creates points then add those points in to points vector. Closes file and returns the vector. Its complexity is $O(n)$ where n is the number of the lines to be read.

```
function main(argc, argv):
    points = readCoordinatesFromFile(argv[1])
    findClosestPairOrder(points)
    return 0
```

Figure 0.7: Main Function

- My main function first calls the readCoordinatesFromFile() function to read the points from the case files. Then it calls findClosestPairOrder() wrapper function to find closest pairs and print on console. Then returns. These functions and their complexities are explained above.

0.2. Questions

1. **What is the time and space complexity of the divide & conquer algorithm?** The time complexity of the divide and conquer algorithm can be expressed with the relation: $T(n) = 2T(n/2) + O(n)$. So that we can solve this relation with master theorem and the time complexity of the divide and conquer algorithm can be found as $O(nlog(n))$ The space complexity of the algorithm is caused by recursive calls and additional spaces for left and right halves and strip vectors. Since the algorithm uses divide and conquer approach, recursive calls take $O(log(n))$ space complexity. Additional vectors take $O(n)$ space complexity. So overall space complexity is $O(n)$.

2. **What is the time and space complexity of the brute force approach?** Time complexity of the brute force approach is $O(n^2)$ since the algorithm iterates over all pairs of points and calculates the distance. The space complexity of the algorithm is $O(1)$. It only uses a constant amount of space for whatever the input is.

3. **Compare the performance of divide & conquer and brute force approaches timewise for each given case. Which one performs better? Why?**

	Case 1	**Case 2**	**Case 3**	**Case 4**	**Case 5**
Brute Force	0.000968 s.	0.0010 s.	0.0028 s.	0.077 s.	1.14095 s.
Divide & Conquer	0.000974 s.	0.0010 s.	0.0031 s.	0.050 s.	0.348083 s.

Table 1: Execution time on different input sizes for brute force and divide and conquer algorithms

As it can be also seen from the table, divide and conquer algorithm performs better but in small-size inputs they may seem close. Since brute-force algorithm is a simpler algorithm it may perform slightly better for very small-sized inputs but overall divide and conquer algorithm has much less execution time since it's recursive algorithm which has $O(nlog(n))$ worst case time complexity compared to brute force algorithm which has $O(n^2)$ time complexity.

4. **Would the results change if we used Manhattan distance instead of Euclidean distance? How?** Since we change the metric from euclidean distance to manhattan distance we have to change the distance calculation function; therefore it would affect both computation and sorting for the points. And we may also need to change sorting and comparing operations for the points. So it would change the results. An example to how it changes the results can be seen below:

   Suppose we have this points:
   Point A: (0, 0)
   Point B: (3, 4)
   Point C: (7, 6)
   Point D: (8, 1)

   For Euclidean distance, the closest pair is (3, 4) and (7, 6) because the distance between them is $\sqrt{(7-3)^2+(6-4)^2} = 5.39$ approximately, which is smaller than all other point pairs.
   However, if we used Manthattan distance:

   Distance between (0, 0) and (3, 4): |0 - 3| + |0 - 4| = 3 + 4 = 7
   Distance between (0, 0) and (7, 6): |0 - 7| + |0 - 6| = 7 + 6 = 13
   Distance between (0, 0) and (8, 1): |0 - 8| + |0 - 1| = 8 + 1 = 9
   Distance between (3, 4) and (7, 6): |3 - 7| + |4 - 6| = 4 + 2 = 6
   Distance between (3, 4) and (8, 1): |3 - 8| + |4 - 1| = 5 + 3 = 8
   Distance between (7, 6) and (8, 1): |7 - 8| + |6 - 1| = 1 + 5 = 6

   We can see that both (3, 4) and (7, 6) pair and, (7, 6) and (8, 1) have the same distance of 6 which is the smallest. So it can be concluded that with different metrics the result changes.