Analysis of Algorithms 2

BLG 336E

Project 2 Report

Abdullah Jafar Mansour Shamout
shamout21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 09.04.2024

1. Closest pair of points

1.1. Complexities over Pseudo-codes

```
FUNCTION distance(p1, p2)://Takes O(1)
    RETURN SquareRoot((p1.x - p2.x)^2 + (p1.y - p2.y)^2)

FUNCTION compareX(p1, p2)://Takes O(1)
    RETURN p1.x < p2.x

FUNCTION compareY(p1, p2)://Takes O(1)
    RETURN p1.y < p2.y

FUNCTION bruteForceClosestPair(points, start, end): //Takes O(n^2)
    min_dist = INFINITY
    closest_pair = empty pair
    FOR i FROM start TO (end - 1): //Takes O(n)
        FOR j FROM (i + 1) TO (end - 1): //Takes O(n)
            dist = distance(points[i], points[j]) //Takes O(1)
            IF dist < min_dist: //Takes O(1)
                min_dist = dist //Takes O(1)
                closest_pair = {points[i], points[j]} //Takes O(1)
    RETURN closest_pair

FUNCTION closestPair(points, start, end): //Complexity: T(n) = 2T(n/2) + O(n) = O(n log n)
    IF end - start <= 3: //Takes O(1)
        RETURN bruteForceClosestPair(points, start, end) //Takes O(n^2)
    mid = (start + end) / 2 //Takes O(1)
    left_pair = closestPair(points, start, mid) //Takes T(n/2)
    right_pair = closestPair(points, mid, end) //Takes T(n/2)
    left_dist = distance(left_pair.first, left_pair.second) //Takes O(1)
    right_dist = distance(right_pair.first, right_pair.second) //Takes O(1)
    IF left_dist < right_dist: //Takes O(1)
        min_pair = left_pair //Takes O(1)
        min_dist = left_dist //Takes O(1)
    ELSE:
        min_pair = right_pair //Takes O(1)
        min_dist = right_dist //Takes O(1)
    strip = empty vector //Takes O(1)
    FOR i FROM start TO end - 1: //Takes O(n)
        IF ABS(points[i].x - points[mid].x) < min_dist: //Takes O(1)
            strip.push_back(points[i]) //Takes O(1)
    sort(strip.begin(), strip.end(), compareY) //Takes O(n log n)
    // The following double loop looks like it takes O(n^2), however due to n staying relatively low with this algorithm it doesnt scale up to n^2 as n increases, it rather stays at a low value thus it can be ignored.
    FOR i FROM 0 TO strip.size() - 1: //Takes O(n)
        FOR j FROM i + 1 TO strip.size() - 1 AND (strip[j].y - strip[i].y) < min_dist: //Takes O(n)
            dist = distance(strip[i], strip[j]) //Takes O(1)
            IF dist < min_dist: //Takes O(1)
                min_dist = dist //Takes O(1)
                min_pair = {strip[i], strip[j]} //Takes O(1)
    RETURN min_pair

FUNCTION removePairFromVector(point_vector, point_pair): //Takes O(n)
    new_vector = empty vector //Takes O(1)
    FOR EACH point IN point_vector: //Takes O(n)
        IF point != point_pair.first AND point != point_pair.second: //Takes O(1)
            new_vector.push_back(point) //Takes O(1)
    RETURN new_vector //Takes O(1)

FUNCTION findClosestPairOrder(points): //Complexity: O(n log n)
    pairs = empty vector of pairs //Takes O(1)
    unconnected = {-1, -1} //Takes O(1)
    sort(points.begin(), points.end(), compareX) //Takes O(n log n)
    closest = closestPair(points, 0, points.size()) //Takes O(n log n)
    WHILE NOT closest.first.x == 0: //Takes O(n)
        IF compareY(closest.first, closest.second) OR (closest.first.y == closest.second.y AND compareX(closest.first, closest.second)): //Takes O(1)
            pairs.push_back(closest) //Takes O(1)
        ELSE:
            pairs.push_back({closest.second, closest.first}) //Takes O(1)
        points = removePairFromVector(points, closest) //Takes O(n)
        closest = closestPair(points, 0, points.size()) //Takes O(n log n)
    IF points.size() == 1: //Takes O(1)
        unconnected = points[0] //Takes O(1)
    FOR i FROM 0 TO pairs.size() - 1: //Takes O(n)
        PRINT "Pair " + (i + 1) + ": " + pairs[i].first.x + ", " + pairs[i].first.y + " - " + pairs[i].second.x + ", " + pairs[i].second.y //Takes O(1)
    IF unconnected.x != -1: //Takes O(1)
        PRINT "Unconnected " + unconnected.x + ", " + unconnected.y //Takes O(1)
```

```
FUNCTION readCoordinatesFromFile(filename): //Takes O(n)
    points = empty vector of points //Takes O(1)
    file = open file(filename) //Takes O(1)
    IF file is open: //Takes O(1)
        WHILE NOT EOF: //Takes O(n)
            line = read line from file //Takes O(1)
            p = empty point //Takes O(1)
            PARSE line and store coordinates in p //Takes O(1)
            points.push_back(p) //Takes O(1)
        CLOSE file //Takes O(1)
    ELSE: //Takes O(1)
        PRINT "File could not be opened." //Takes O(1)
    RETURN points //Takes O(1)

FUNCTION main(argc, argv): //Takes O(n log n)
    points = readCoordinatesFromFile(argv[1]) //Takes O(n)
    findClosestPairOrder(points) //Takes O(n log n)
    RETURN 0 //Takes O(1)
```

1.2. Questions and Answers

1.2.1. Time and space complexities for greedy algorithm

The time complexity of the greedy algorithm (divide and conquer) used for finding the closest pair of points can be analyzed as follows:

- The time complexity of sorting the points based on their x-coordinates is $O(n \log n)$, where n is the number of points.
- The time complexity of the divide and conquer algorithm is $T(n) = 2T(n/2) + O(n)$, which results in $O(n \log n)$ time complexity.
- Therefore, the overall time complexity of the greedy algorithm is $O(n \log n)$.

The space complexity of the greedy algorithm mainly depends on the space required for recursion stack and additional data structures. Since the recursion depth is $O(\log n)$ in the worst case, the space complexity of the algorithm is $O(\log n)$. Keep in mind that I am ignoring other variables created during the running of the code; such as the strip vector, the new vectors after removing pairs, point variables and such. If I am to count them then it would be $O(n)$ since the vector that I use to store the points in takes n points.

1.2.2. Time and space complexities for brute force algorithm

The time complexity of the brute force algorithm for finding the closest pair of points can be analyzed as follows:

- The brute force algorithm compares all possible pairs of points, resulting in $O(n^2)$ comparisons, where n is the number of points.
- For each comparison, the distance between two points is calculated, which takes $O(1)$ time.
- Therefore, the overall time complexity of the brute force algorithm is $O(n^2)$.

The space complexity of the brute force algorithm mainly depends on the space required to store the points and the closest pair found. Since no additional data structures are used, the space complexity is $O(1)$ in terms of the input size n.

1.2.3. Results

Table 1.1 presents the comparison of execution times for the divide and conquer approach and the brute force approach in nanoseconds for different cases.

	Case 0	**Case 1**	**Case 2**	**Case 3**	**Case 4**
Divide and Conquer	68016ns	42957ns	207983ns	796387ns	1244226ns
Brute Force	7541ns	17820ns	848045ns	13573281ns	55142303ns

Table 1.1: Comparison of execution times for the divide and conquer and brute force approaches per case

1.2.4. Would the results change if we used Manhattan distance instead of Euclidean distance? How?

Yes, the results would likely change if Manhattan distance were used instead of Euclidean distance. The main difference between these distance metrics is how they measure distance between points. Euclidean distance is the straight-line distance between two points in a plane, while Manhattan distance is the sum of the horizontal and vertical distances between two points along grid lines.

Since the algorithms rely on distance calculations between points, the change in distance metric would affect the distance comparisons made within the algorithms. Manhattan distance tends to yield different distance values compared to Euclidean distance for the same pair of points, which can result in different pairs of points being identified as the closest. Therefore, the execution times for the algorithms would likely vary when using Manhattan distance instead of Euclidean distance, as the computational workload and comparisons would differ.