Analysis of Algorithms II

BLG 336E

Project 2 Report

Mustafa Can Çalışkan
caliskanmu20@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering

1. Implementation

1.1. Pseudo-Code

The pseudocodes and their explanations of the source codes are provided below.

1.1.1. Helper Functions

```
function distance(p1, p2)
    return √((p2.x − p1.x)² + (p2.y − p1.y)²)
end function
function compareX(p1, p2)
    return p1.x < p2.x
end function
function compareY(p1, p2)
    return p1.y < p2.y
end function
```

The time complexity of these functions is $O(1)$ as they only perform control and distance calculations for sorting.

1.1.2. bruteForceClosestPair

In this function, the pair of points with the minimum distance within the provided interval is determined using the method of traversing through all other points for each point, employing the brute force technique.

Algorithm 1 Brute-Force Closest Pair

```
function bruteForceClosestPair(points, start, end)
    closest_result ← empty pair of points
    min_dist ← maximum double value
    for i ← start to end − 1 do
        for j ← i + 1 to end do
            dist ← distance(points[i], points[j])
            if dist < min_dist then
                min_dist ← dist
                closest_result ← {points[i], points[j]}
            end if
        end for
    end for
    return closest_result
end function
```

Since each point is visited for all other points, the total time complexity is $O(n^2)$, where n represents the number of points.

1.1.3. closestPair

In this function, the well-known divide & conquer method is utilized where the given range is recursively divided into two halves until reaching the base case, at which point the brute force method is employed. Additionally, points close to the division boundary are also separately examined in the function. As a result, the function returns the pair of points with the minimum distance within the given range.

Algorithm 2 Closest Pair Algorithm

```
function closestPair(points, start, end)
    if end − start ≤ 3 then
        return bruteForceClosestPair(points, start, end)
    end if
    middle ← (start + end)/2
    left_pair ← closestPair(points, start, middle)
    right_pair ← closestPair(points, middle, end)
    closest ← empty pair of points
    min_dist ← distance(left_pair.first, left_pair.second)
    if distance(right_pair.first, right_pair.second) < min_dist then
        closest ← right_pair
        min_dist ← distance(right_pair.first, right_pair.second)
    end if
    strip ← empty list of points
    for i ← start to end do
        if |points[i].x − points[middle].x| < min_dist then
            strip.push_back(points[i])
        end if
    end for
    sort(strip.begin(), strip.end(), compareY)
    for i ← 0 to strip.size() − 1 do
        for j ← i + 1 to strip.size() do
            if strip[j].y − strip[i].y < min_dist then
                dist ← distance(strip[i], strip[j])
                if dist < min_dist then
                    min_dist ← dist
                    closest ← {strip[i], strip[j]}
                end if
            else
                break
            end if
        end for
    end for
    return closest
end function
```

The time complexity of this function is $O(nlogn)$ because it is implemented using a divide and conquer algorithm. The function divides itself into halves and solves these subproblems to find the closest pair for each divided part. The division process occurs in $O(logn)$ steps. The brute force part operates in at most $O(1)$ steps since it handles at most 3 points, resulting in a constant workload. To calculate the distance between the closest pairs, each subproblem needs to compare the closest two pairs in $O(n)$ steps. Consequently, the divide and conquer part has a complexity of $O(logn)$, and each subproblem's solution has a complexity of $O(n)$. Therefore, the overall time complexity is $O(nlogn)$.

1.1.4. removePairFromVector

In this function, the identified closest pair of points is located within the points vector and subsequently removed.

Algorithm 3 Remove Pair From Vector

function removePairFromVector(point_vector, point_pair)
    result_vector $\leftarrow$ empty vector of points
    for each point in point_vector **do**
        if not (($point.x = point\_pair.first.x$ and $point.y = point\_pair.first.y$) or ($point.x = point\_pair.second.x$ and $point.y = point\_pair.second.y$)) **then**
            result_vector.push_back(point)
        end if
    end for
    return result_vector
end function

The time complexity of the function is $O(n)$ since it involves searching for and removing the elements of the given pair in the vector, and this operation depends on the number of elements in the vector, denoted by n.

1.1.5. findClosestPairOrder

In this function, the closestPair function is invoked in each iteration until the points vector is emptied. During each iteration, the function finds the pair of points with the shortest distance, adhering to the sequence specified in the assignment description. These pairs are then appended to the result vector. Consequently, pairs of points with the shortest distance are obtained.

Algorithm 4 Find Closest Pair in Order

```
procedure findClosestPairOrder(points)
    pairs ← empty vector of pairs
    unconnected ← Point (−1, −1)
    sort(points.begin(), points.end(), compareX)
    while ¬points.empty() and points.size() ≠ 1 do
        closest ← closestPair(points, 0, points.size())
        if (closest.first.y > closest.second.y) or (closest.first.y = closest.second.y and closest.first.x > closest.second.x) then
            swap(closest.first, closest.second)
        end if
        pairs.push_back(closest)
        points ← removePairFromVector(points, closest)
    end while
    if ¬points.empty() then
        unconnected ← points[0]
    end if
    for i ← 0 to pairs.size() − 1 do
        Print Pairs
    end for
    if unconnected.x ≠ −1 then
        Print Unconnected
    end if
end procedure
```

The function has a time complexity of $O(n^2 logn)$ since it calls the closestPair function for each element of the points vector, and all other operations ($O(n)$ for removePairFromVector, $O(nlogn)$ for sorting, $O(n)$ for printing, etc.) have a complexity smaller than $O(n^2 logn)$.

2. Discussions

1. **What is the time and space complexity of the divide & conquer algorithm?**

   As mentioned above, since the divide and conquer algorithm continuously divides the map into halves and performs subtasks in each part, its time complexity becomes $O(nlogn)$.

2. **What is the time and space complexity of the brute force approach?**

   Again, as mentioned above, when using the brute force algorithm, each point within the range needs to be checked against all other points, resulting in a time complexity of $O(n^2)$.

3. **Comparison of the performance of divide & conquer and brute force approaches**

   The performance analysis of different approaches is provided below.

Cases	Divide & Conquer	Brute Force
Case 0	143148	123578
Case 1	174958	163677
Case 2	4903173	7317731
Case 3	57792938	152701273
Case 4	208302990	1930244857

Table 2.1: Performance Analysis of Different Approaches (Measurement Unit: Nanoseconds)

   As predicted by our time complexity analysis, while approaches may be close to each other in small-sized maps, as the map size increases, the divide & conquer approach significantly outperforms brute force in terms of performance.

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

   Since only the computation type in the distance function will change, and both computation types have a time complexity of O(1), the change in distance type will not directly result in a difference in time complexity. However, due to the ease of computing Manhattan distance, albeit marginally, there might be a slight improvement in performance.