Analysis of Algorithms II

BLG 336E

Project 2 Report

Rengin Helin Yalçın
yalcinr22@itu.edu.tr

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

1. Implementation

1.1. Implementation of the Algorithms

- The code implements an algorithm to find the closest pair of points among a set of 2D points using divide and conquer approach, with a fallback to brute force for small sets.

- While the brute force approach is not the most efficient solution for finding the closest pair of points in general, it provides a simple and effective fallback strategy for handling small datasets in the context of the divide and conquer algorithm implemented in the code.

	Map 1	**Map 2**	**Map 3**	**Map 4**	**Map 5**
Divide-Conquer	113300 ns	114700 ns	3480900 ns	39387100 ns	248763000 ns
Brute Force	78100 ns	110100 ns	5208000 ns	247394500 ns	3647342100 ns

Table 1.1: Comparison on different maps.

- In general, the divide-and-conquer algorithm performs better in terms of execution time, especially on larger and more complex maps. However, for smaller maps with fewer points, the brute force approach may be slightly faster due to the overhead of recursion and sorting in the divide-and-conquer algorithm, as observed in maps 1 and 2.

- The test results confirm the theoretical expectations regarding the performance of both algorithms. The divide-and-conquer algorithm provides a more scalable and efficient solution for finding the closest pair of points, especially for larger datasets, while the brute force approach serves as a simpler but less efficient alternative,

1.1.1. Manhattan Distance

- Manhattan distance measures the distance between two points by summing the absolute differences between their coordinates while Euclidean distance is calculated as the square root of the sum of the squared differences between the coordinates of the points.

- Tables below compare the performance of Euclidean distance and Manhattan distance calculations using two different approaches: divide and conquer, and brute force.

	Map 1	Map 2	Map 3	Map 4	Map 5
Euclidean	113300 ns	114700 ns	3480900 ns	39387100 ns	248763000 ns
Manhattan	375100 ns	146700 ns	2406700 ns	37955400 ns	254171800 ns

Table 1.2: Comparison of performance with divide and conquer.

	Map 1	Map 2	Map 3	Map 4	Map 5
Euclidean	78100 ns	110100 ns	5208000 ns	247394500 ns	3647342100 ns
Manhattan	134900 ns	98100 ns	1788200 ns	60783700 ns	875261400 ns

Table 1.3: Comparison of performance with brute force approach.

- In the first table, both Euclidean and Manhattan distance calculations show an increase in time as the complexity of the map increases. The difference between the performances gets less observable as the complexity of the maps increase yet it can still be said that Manhattan performs slightly better.

- Similarly, in the second table, as the complexity of the maps increases, both Euclidean and Manhattan distances experience an increase in time. However, as the complexity of the maps increases, Manhattan approach shows much more efficient results.

1.2. Code

- The time complexity of the `bruteForceClosestPair` function is $O(n^2)$, where n is the number of points in the input vector. This is because there are two nested loops that iterate through all possible pairs of points in the input vector. The space complexity of the function is $O(1)$ because it only uses a constant amount of extra space regardless of the size of the input vector.

- The time complexity of the `closestPair` function can be expressed as $T(n) = 2T(n/2) + O(n)$. By using the Master Theorem, we can conclude that the time complexity of the `closestPair` function is $O(nlogn)$.

  As the function is recursive, space is allocated on the stack for each recursive call. The maximum depth of recursion in this function is logarithmic in terms of the number of points, which means it's $O(logn)$.The algorithm may use additional space for storing intermediate data, such as the strip of points sorted by y-coordinate. The maximum size of this data structure is proportional to the number of points, so it's $O(n)$. The total space complexity of the closestPair function is $O(n)$ due to the dominant factor being the auxiliary data structures.

Algorithm 1 Divide and Conquer Algorithm

```
function closestPair(points, start, end)
    if end − start ≤ 3 then
        return bruteForceClosestPair(points, start, end)
    end if
    mid ← (start + end)/2
    leftPair ← closestPair(points, start, mid)
    rightPair ← closestPair(points, mid, end)
    leftDist ← distance(leftPair.first, leftPair.second)
    rightDist ← distance(rightPair.first, rightPair.second)
    minPair
    minDist
    if leftDist < rightDist then
        minPair ← leftPair
        minDist ← leftDist
    else
        minPair ← rightPair
        minDist ← rightDist
    end if
    strip
    for i ← start to end do
        if |points[i].x − points[mid].x| < minDist then
            push_back(strip, points[i])
        end if
    end for
    sort(strip.begin(), strip.end(), compareY)
    for i ← 0 to strip.size() do
        for j ← i + 1 to strip.size() and (strip[j].y − strip[i].y) < minDist do
            dist ← distance(strip[i], strip[j])
            if dist < minDist then
                minDist ← dist
                minPair ← {strip[i], strip[j]}
            end if
        end for
    end for
    return minPair                                  ▷ Time Complexity: O(nlogn)
end function
```

Algorithm 2 Brute Force Algorithm

function bruteForceClosestPair($points, start, end$)
    $closest$
    $minDist \leftarrow$ numeric_limits<double>::max()
    for $i \leftarrow start$ **to** end **do**
        for $j \leftarrow i + 1$ **to** end **do**
            $dist \leftarrow$ distance($points[i], points[j]$)
            if $dist < minDist$ **then**
                $minDist \leftarrow dist$
                $closest \leftarrow \{points[i], points[j]\}$
            end if
        end for
    end for
    return $closest$ $\triangleright$ Time Complexity: $O(n^2)$
end function