Analysis of Algorithms II

BLG 336E

Project 2 Report

KAAN KARATAŞ
karatask20@itu.edu.tr

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

1. Implementation

The project is essentially an implementation of an algorithm for finding the closest connections between points in a 2D space, as stated in the homework assignment. It is structured to read a set of coordinates from a text file, execute either a brute-force or a divide-and-conquer approach depending on user choice, and subsequently display the top results. The program accepts a single command-line argument to specify coordinates of individual cities. The program also measures the time taken to perform the traversal operation. This traversal time serves as a benchmark for comparisons among different algorithms implemented within the project.

1.1. Point Struct And Helper Functions

1.1.1. Point

The code uses a structure named ‘Point‘ to represent 2D points, consisting of ‘x‘ and ‘y‘ coordinates. Equality operator is overloaded to enable more straightforward comparison between two points based on coordinates they posses.

Algorithm 1 Point Structure Definition

```
1: struct Point {
2:         double x, y;
3:
4:         // Equality operator overloading to compare two points) {
5:         bool operator==(const Point& other) const {
6:             return x == other.x and y == other.y;
7:         }
8: };
```

1.1.2. Distance

The ’distance’ function computes the Euclidean distance between two points in a two-dimensional space using the standard distance formula. It takes two points, p1 and p2, as input and returns their distance as a double value.

Algorithm 2 Calculate Euclidean Distance

1: **function** distance(Point $p1$, Point $p2$)
2: **return** $\sqrt{(p1.x - p2.x)^2 + (p1.y - p2.y)^2}$ ▷ Euclidean distance
3: **end function**

1.1.3. Compare X

The 'compareX' function is a helper function used to compare two points based on their x-coordinates. It takes two points, p1 and p2, as input and returns true if the x-coordinate of p1 is less than that of p2, indicating that p1 should precede p2 in the sorting order.

Algorithm 3 Compare Points based on X-coordinate

```
1: function compareX(Point p1, Point p2)
2:     return p1.x < p2.x                ▷ Sort based on x-coordinate
3: end function
```

1.1.4. Compare Y

The 'compareY' function serves as a helper function for sorting points based on their y-coordinates. Given two points, p1 and p2, it returns true if the y-coordinate of p1 is less than that of p2, indicating that p1 should come before p2 in the sorted order.

Algorithm 4 Compare Points based on Y-coordinate

```
1: function compareY(Point p1, Point p2)
2:     return p1.y < p2.y                ▷ Sort based on y-coordinate
3: end function
```

1.2. Brute Force Approach For Calculating Closest Pairs

Brute force, in the context of finding the closest pair of points, involves tirelessly examining every possible pair and calculating the distance between them to determine the closest pair. The bruteForceClosestPair function provided below, shows an example of this approach.

The function takes a vector of points along with the start and end indices to define the subset of points to analyze. It initializes the closest pair and the minimum distance using the first two points in the given subset.

Then, nested loops iterate over all pairs of points within the specified range. For each pair, the distance between is calculated using the 'distance' helper function. If the calculated distance is smaller than the current minimum distance, both the minimum distance and the closest pair are updated accordingly.

Algorithm 5 Brute Force Closest Pair Algorithm

```
1:  function BruteForceClosestPair(points, start, end)
2:      closestPair ← pair<Point, Point>                    ▷ Initialize the closest pair
3:      minDistance ← distance(points[start], points[start + 1])    ▷ initial min distance
4:      for i ← start to end − 1 do
5:          for j ← i + 1 to end do
6:              dist ← distance(points[i], points[j])   ▷ Calculate distance between points
7:              if dist < minDistance then
8:                  minDistance ← dist                  ▷ Update minimum distance
9:                  closestPair ← points[i], points[j]  ▷ Update closest pair
10:             end if
11:         end for
12:     end for
13:     return closestPair                              ▷ Return the closest pair
14: end function
```

1.2.1. Complexity Analysis

The time complexity of this brute force approach is $O(n^2)$, with n being the number of points in the given subset. This is because the algorithm iterates through all possible pairs of points within the specified range, resulting in nested loops that each iterate through n elements. Therefore, the time complexity of the function is $O(n^2)$. Despite its simplicity, this approach can become inefficient for large datasets due to its quadratic time complexity.

The space complexity of the brute force approach implemented in the function ‘bruteForceClosestPair‘ is $O(1)$. This is because the algorithm only uses a constant amount of additional memory regardless of the size of the input. It does not allocate any extra space proportional to the input size n, making its space complexity constant. Therefore, the space complexity of the function is $O(1)$.

1.2.2. To Use Brute Force, Or Not To Use Brute Force?

This straightforward brute force approach ensures that every possible pair of points is considered, making it a reliable method for finding the closest pair. However, its time complexity grows quadratically with the number of points, making it less efficient for large datasets compared to more sophisticated algorithms like the divide-and-conquer approach.

1.3. Divide & Conquer Approach For Calculating Closest Pairs

The provided function implements a divide and conquer algorithm to find the closest pair of points in a set of 2D points. The main objective of this algorithm is to recursively divide the set of points into smaller subsets until a base case is reached, and then combine

the results to find the closest pair.

The function 'closestPair' begins by checking if there are 3 or fewer points in the current subset. If so, it delegates the task of finding the closest pair to the bruteForceClosestPair function, which exhaustively compares all pairs of points within the subset.

If there are more than 3 points, the function recursively divides the set into two halves and computes the closest pairs for each half. It then determines the minimum distance between the closest pairs from the left and right halves.

To handle the case where the closest pair consists of points from both halves, the function constructs a "strip" of points centered around the middle point of the subset. It then iterates over the points in this strip and compares their distances to each other to find the closest pair.

Overall, the divide and conquer approach significantly reduces the number of point comparisons compared to brute force, leading to improved efficiency, especially for large datasets.

1.3.1. Complexity Analysis

The time complexity of the divide and conquer approach is $O(nlogn)$, where n is the number of points in the given subset. This complexity is a result of the recursive logic of the algorithm. In each recursive call, the points are divided into two halves, and the algorithm recursively finds the closest pairs in each half. Then, results are combined in linear time. The overall time complexity can be expressed as $T(n) = 2T(n/2) + O(n)$, which results in $O(nlogn)$ time complexity.

The space complexity of the divide and conquer approach is $O(n)$, where n is the number of points in the given subset. This complexity is a result of the recursive function calls and the additional space required for storing the strip of points. In each recursive call, additional space is allocated for function call stacks. Additionally, the strip vector used to store points within the minimum distance from the middle point contributes to the space complexity. Thus, the space complexity of the function is $O(n)$.

Algorithm 6 Divide And Conquer Closest Pair Algorithm

```
function ClosestPair(points, start, end)
    if end - start ≤ 3 then                ▷ Base case: Use brute force for small point sets
        return BruteForceClosestPair(points, start, end)
    end if
    mid ← (start + end)/2                                          ▷ Find the middle point
    midPoint ← points[mid]                                         ▷ Get the middle point
    leftPair ← ClosestPair(points, start, mid)                     ▷ Recursive left side
    rightPair ← ClosestPair(points, mid, end)                      ▷ Recursive right side
    leftDistance ← Distance(leftPair.first, leftPair.second)
    rightDistance ← Distance(rightPair.first, rightPair.second)
    minDistance ← min(leftDistance, rightDistance)                 ▷ Find the minimum distance
    closestPair ← (leftDistance < rightDistance) ? leftPair : rightPair
    strip ← empty vector                ▷ Create a vector to store the points within the strip
    for each point p in points[start:end] do
        if abs(p.x − midPoint.x) ≤ minDistance then
            strip.push_back(p)
        end if
    end for
    Sort(strip, compareY)                          ▷ Sort the strip vector based on y-coordinate
    for each point p in strip do
        for each remaining point q in strip do
            if (q.y − p.y) < minDistance then                      ▷ Check if points are within
                dist ← Distance(p, q)                              ▷ Calculate the distance between
                if dist < minDistance then                         ▷ Update if distance is less than
minDistance
                    minDistance ← dist
                    closestPair ← (p, q)
                end if
            end if
        end for
    end for
    return closestPair                                             ▷ Return the closest pair
end function
```

1.3.2. To Use Divide and Conquer, Or Not To Use Divide and Conquer?

The divide-and-conquer algorithm offers a more efficient alternative to brute force for finding the closest pair of points. By recursively dividing the point set into smaller subsets and solving them independently before combining the results, this approach significantly reduces the number of comparisons needed. This leads to a lower time complexity, particularly for larger datasets, compared to the brute force method. However, the divide-and-conquer approach may incur some overhead due to the recursive calls and the additional steps required to merge the results. Nevertheless, its overall efficiency

makes it a preferred choice for many applications where performance is crucial.

1.4. Other Functions

1.4.1. Removing Pairs

The ’removePairFromVector’ function removes a given pair of points from the provided point vector. The ’erase’ and ’remove’ functions from the srandart library are used. After the points are removed, the updated vector containing the remaining points is returned. The aim of this function is to ensure that the specified pair of points are excluded from repetitive consideration.

Algorithm 7 Remove Pair From Vector

```
1: function removePairFromVector(vector, p_pair)
2:     Find and remove the first point of the pair from  vector
3:     vector.erase(remove(vector.begin(), vector.end(), p_pair.first), vector.end())
4:     Find and remove the second point of the pair from  vector
5:     vector.erase(remove(vector.begin(), vector.end(), p_pair.second), vector.end())
6:     return vector
7: end function
```

1.4.2. Reading Input From File

The ’readCoordinatesFromFile’ function is there to extract coordinates from a given file and convert those coordinates into a vector of ’Point’ structs. At first, the function initializes an empty vector to store points and tries to open the specified file. If the file is successfully opened, each line is iterated through, parsing the coordinates and creating a ’Point’ object for each line. These points are then added to the vector. Finally, the function returns the vector containing all the points taken from the file. If the file can not be opened, an empty vector is returned.

Algorithm 8 Read Coordinates From File

```
1: function readCoordinatesFromFile(filename)
2:     points ← empty vector                                   ▷ Vector to store the Points
3:     file ← open(filename)                                   ▷ Open file
4:     if ¬file.is_open() then
5:         return points          ▷ Return an empty vector if the file cannot be opened
6:     end if
7:     line ← empty string
8:     while getline(file, line) do                            ▷ Read each line from the file
9:         p ← new Point                                       ▷ Create a new Point
10:        sscanf(line.c_str(), "%lf %lf", &p.x, &p.y) ▷ Parse the line to extract the x and
   y coordinates
11:        points.push_back(p)                                 ▷ Add the Point to the vector
12:    end while
13:    file.close()                                            ▷ Close file
14:    return points                                           ▷ Return vector of Points
15: end function
```

1.4.3. Main Logic Function

The ’findClosestPairOrder’ function is the main function of implemented algorithms that finds the closest pairs of points. It takes a vector of ‘Point‘ objects as input. Initially, it sorts the points based on their x-coordinates. Then, it iterates through the points, finding the closest pair using either one of the ipmlemented approaches explained above. The closest pairs are stored in the ’pairs’ vector, and the paired points are removed from the original vector at each iteration.

If any points remain unpaired, the function sets the ’unconnected’ var accordingly. Before the output is printed, the function ensures that within each pair, the city with the smaller y-coordinate is printed first, and then checks the x-coordinate to print in correct order. Finally, the pairs are printed and the unconnected points are printed as well providing they exist.

Algorithm 9 Find Closest Pair Order

```
1: function findClosestPairOrder(points)
2:     pairs ← empty vector
3:     unconnected ← Point(−1, −1)
4:     sort(points.begin(), points.end(), compareX)          ▷ Sort the points based on x-coordinate
5:     while points.size() > 1 do
6:         closest ← bruteForceClosestPair(points, 0, points.size())  ▷ Find the closest pair using brute force
7:         pairs.push_back(closest)                          ▷ Add the closest pair to the vector
8:         removePairFromVector(points, closest)             ▷ Remove paired points from vector
9:     end while
10:    if ¬points.empty() then
11:        unconnected ← points[0]                           ▷ If one point left, it remains unconnected
12:    end if
13:    for all p ∈ pairs do                                  ▷ Sort based on: smaller y first, then smaller x
14:        if p.first.y > p.second.y or (p.first.y == p.second.y and p.first.x > p.second.x) then
15:            swap(p.first, p.second)
16:        end if
17:    end for
18:    for i ← 0 to pairs.size() do
19:        print("Pair " + (i+1) + ": " + pairs[i].first.x + ", " + pairs[i].first.y)
20:        print(" - " + pairs[i].second.x + ", " + pairs[i].second.y)
21:    end for
22:    if unconnected.x ≠ −1 then
23:        print("Unconnected " + unconnected.x + ", " + unconnected.y)
24:    end if
25: end function
```

1.5. Extra Comments

1.5.1. Time Comparison Between Approaches

Comparing the execution times of divide & conquer and brute force approaches across datasets can give important information about their performance. For smaller datasets (such as Maps 1 and 2), the brute force approach tends to be quicker. However, as the dataset size grows (like in the cases of Maps 3 and 4), the divide and conquer approach demonstrates superior performance. This difference is a result of the brute force approach's quadratic time complexity, making it less efficient for larger datasets compared to the divide and conquer approach, which has a more desirable time complexity closer to $O(nlogn)$. Thus, while the brute force approach may be effective and sufficient for smaller datasets, the divide and conquer approach clearly excels for larger ones.

	Map 1	Map 2	Map 3	Map 4
Brute Force	0.001773 ms	1.9651 ms	88.1471 ms	1346.39 ms
Divide & Conquer	0.11741 ms	1.39964 ms	25.263 ms	153.152 ms

Table 1.1: Comparison of different pivoting strategies on input data in milliseconds.

1.5.2. Manhattan Distance Instead Of Euclidean

Manhattan distance measures the distance between two points by summing the absolute differences in their coordinates, which is quite different from the Euclidean distance formula. This change of method would certainly effect the speed of the program as the Euclidean Distance formula consists of square root and power of 2 operations. As these operations take time to complete in C++, a more mathematically and programmatically simpler approach such as the Manhattan Distance would effectively yield in faster results. However, a change of complexity in either of the algorithms is not expected as this distance calculation method does not directly affect the complexities of the functions.