Analysis of Algorithms II

BLG 336E

Assignment 2 Report

Yusuf Yıldız
yildizyu21@itu.edu.tr

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

1. Implementation Details

1.1. Divide&Conquer Algorithm

1.1.1. Time Complexity Analysis

The provided closestPair function implements the closest pair of points algorithm using a divide and conquer approach with a brute force optimization for small subsets. Let's break down the time complexity:

Divide and Conquer Steps:

- Base Case ($O(1)$): If there are 3 or fewer points (end $-$ start ≤ 3), the function calls `bruteForceClosestPair` which has a constant time complexity (discussed later).
- Divide ($O(1)$): Dividing the points into two halves takes constant time (`mid_ix` calculation).
- Conquer ($T(n/2) + T(n/2)$): The function recursively calls itself on the left and right halves, each with roughly $n/2$ points. This leads to a term of $2 \times T(n/2)$ in the time complexity.
- Merge ($O(n)$): Finding the minimum distance and closest pair among the left and right subproblems takes linear time ($O(n)$) due to comparisons and finding the minimum.
- Strip Processing ($O(n) + T'(n')$):
  - Finding points within the strip takes linear time ($O(n)$) in the worst case.
  - If there are less than 2 points in the strip (constant time check), it doesn't call `bruteForceClosestPair`. Otherwise, it calls `bruteForceClosestPair` on a smaller set (`points_on_strip`) with size n'. This introduces another term $T'(n')$.

Time Complexity:

Putting it all together, the time complexity can be represented by the following recurrence relation:

$$T(n) = 2 \times T(n/2) + O(n) + T'(n')$$

where:

- $T(n)$ is the time complexity for n points.
- $T'(n')$ is the time complexity for points within the strip ($n' \leq n$).

Simplifying the Recurrence:

In the worst case, the size of the strip (n') is proportional to n. This means $T'(n')$ can be approximated as $O(n)$.

Using the Master Theorem (assuming a constant work factor in the merge step), we can classify this recurrence as a divide-and-conquer with a linear work factor (divide and conquer with $O(n)$ work at each level).

The Master Theorem tells us that for such a recurrence, the time complexity is:

$$T(n) = O(n \log n)$$

Brute Force Time Complexity (within closestPair):

The `bruteForceClosestPair` function uses nested loops to compare all pairs of points. In the worst case, it performs $\binom{n}{2}$ comparisons (n choose 2), which is approximately $O(n^2)$.

Overall Time Complexity:

The dominant term in the `closestPair` function is the divide and conquer approach with a time complexity of $O(n \log n)$. The brute force calls within the strip processing contribute a smaller factor in most cases. Therefore, the overall time complexity of the `closestPair` algorithm for finding the closest pair of points is $O(n \log n)$.

1.1.2. Space Complexity Analysis

The space complexity of the closestPair algorithm refers to the additional memory it uses besides the input data itself. Here's a breakdown:

- Recursive Calls: The divide-and-conquer approach leads to recursive calls for the left and right halves of the points. In the worst case, the depth of recursion can reach $\log n$ (number of times the data gets divided by 2). Each recursive call creates a new stack frame with some constant amount of space for local variables and function parameters. This contributes a space complexity of $O(\log n)$ due to the constant space per call and logarithmic number of calls.

- Temporary Variables: Within the function, there are temporary variables used for calculations like mid_ix, min_distance, left_pair, etc. These variables typically hold constant or a small amount of data related to the input size. Their space complexity can be considered $O(1)$.

- points_on_strip: In the strip processing step, a temporary vector points_on_strip is used to store points within a specific distance of the middle point. In the worst case, all points can be within the strip, leading to a space complexity of $O(n)$ for this vector.

Overall Space Complexity:

Considering the dominant factors, the space complexity of the closestPair algorithm is determined by the recursive calls ($O(\log n)$) and the points_on_strip vector ($O(n)$). In most cases, the number of points (n) will be larger than the logarithmic term, so the overall space complexity can be approximated as $O(n)$. The detailed pseudo-code of the Divide&Conquer algorithm is as follows:

```
closestPair(points, start, end):
  if end − start ≤ 3:                                                                  O(1)
    return bruteForceClosestPair(points, start, end)                                   O(1)
  mid_ix ← (start + end) / 2                                                           O(1)
  mid_point ← points[mid_ix]                                                           O(1)
  left_pair ← closestPair(points, start, mid_ix)                                       T(n/2)
  right_pair ← closestPair(points, mid_ix, end)                                        T(n/2)
  left_closest_distance ← distance(left_pair.first, left_pair.second)                  O(1)
  right_closest_distance ← distance(right_pair.first, right_pair.second)               O(1)
  min_distance ← min(left_closest_distance, right_closest_distance)                    O(1)
  minimum_pair ← (left_closest_distance < right_closest_distance) ? left_pair : right_pair   O(1)
  points_on_strip ← []                                                                 O(1)
  for i from start to end:                                                             O(n)
    if abs(points[i].x - mid_point.x) < min_distance:                                  O(1)
      append points[i] to points_on_strip                                              O(1)
  if length(points_on_strip) < 2:                                                      O(1)
    return minimum_pair                                                                O(1)
  closest_strip_pair ← bruteForceClosestPair(points_on_strip, 0, length(points_on_strip))   T'(n')
  if distance(closest_strip_pair.first, closest_strip_pair.second) < min_distance:     O(1)
    return closest_strip_pair                                                          O(1)
  return minimum_pair                                                                  O(1)
```

1.2. Brute-Force Algorithm

1.2.1. Time Complexity Analysis

The provided `bruteForceClosestPair` function implements a simple brute force approach to find the closest pair of points in a given set. Let's analyze its time complexity:

Algorithm Steps:

- **Initialization ($O(1)$):** Setting `min_distance` and initializing `closest_pair` take constant time.
- **Nested Loops ($O(n^2)$):** The function uses nested loops to iterate through all pairs of points. The outer loop iterates from `start` to `end-1` (n elements). For each iteration of the outer loop, the inner loop iterates from $i + 1$ to `end-1` ($n - i - 1$ elements). In the worst case, the total number of iterations becomes: $n \times (n - 1)/2 \approx n^2/2$ (ignoring constant factors).
- **Distance Calculation ($O(1)$):** Within each iteration of the inner loop, the function calculates the distance between two points using a function assumed to have constant time complexity (e.g., Euclidean distance formula).

Time Complexity:

The dominant factor in the time complexity comes from the nested loops, resulting in:

$$T(n) = O(n^2 \times 1) = O(n^2)$$

Explanation:

- The constant time operations like initialization and distance calculation contribute a negligible amount compared to the nested loops.
- The number of comparisons and calculations grows quadratically with the number of points (n).

Overall Time Complexity:

The `bruteForceClosestPair` algorithm has a time complexity of $O(n^2)$. It becomes inefficient for large datasets due to the significant increase in comparisons as the number of points grows. This is why the divide-and-conquer approach used in `closestPair` is preferred for larger datasets as it achieves a more favorable $O(n \log n)$ time complexity.

1.2.2. Space Complexity Analysis

The space complexity of the provided `bruteForceClosestPair` function is dominated by the constant factors and can be considered $O(1)$.

Here's a breakdown of the space used:

Constant size variables:

- `min_distance`: Stores the minimum distance found so far and is a primitive data type (`double`) with constant space usage.
- `closest_pair`: Stores the current closest pair of points as a pair of Points. In most implementations, a Point itself likely uses constant space for its x and y coordinates.

Loop iterators:

- `i` and `j`: These are integer iterators for the loops, and integers typically have constant space requirements.

Overall Space Complexity:

There are no additional data structures created within the loops that scale with the input size. Therefore, the space complexity is dominated by the constant size variables and loop iterators, resulting in $O(1)$. The detailed pseudo-code of the Brute-Force algorithm is as follows:

```
bruteForceClosestPair(points, start, end):
  min_distance ← numeric_limits<double>::max()                          O(1)
  closest_pair ← {points[start], points[start + 1]}                     O(1)
  for i from start to end − 1:                                          O(n^2)
    for j from i + 1 to end − 1:                                        O(n)
      current_distance ← distance(points[i], points[j])                 O(1)
      if current_distance < min_distance:                               O(1)
        min_distance ← current_distance                                 O(1)
        closest_pair ← {points[i], points[j]}                           O(1)
  return closest_pair                                                   O(1)
```

1.3. removePairFromVector Algorithm

1.3.1. Time Complexity Analysis

Here's a breakdown of the factors contributing to the complexity of removePairFromVector algorithm:

- **Loop Iteration (O(n))**: The for loop iterates through each element (point) in the `point_vector`. In the worst case, this loop runs n times (once for each element).
- **Conditional Check (O(1))**: Inside the loop, there's a conditional check to see if the current point matches either element of the `point_pair` to be removed. This comparison involves constant time operations and doesn't affect the overall complexity.
- **Appending to New Vector (O(1))**: If the point doesn't match the elements to be removed, it's appended to the `new_vector` using `push_back`. In most vector implementations, `push_back` has an amortized constant time complexity, meaning adding individual elements is efficient.

Overall Time Complexity:

Since the dominant factor is the loop iterating through n elements, and all operations inside the loop have constant time complexity, the overall time complexity of removePairFromVector is $O(n)$.

1.3.2. Space Complexity Analysis

The space complexity of the removePairFromVector function is $O(n)$, where n is the number of elements in the original point_vector.

Here's the breakdown of the space requirements:

- **New Vector (O(n))**: The function creates a new vector `new_vector` to store the elements that are not part of the `point_pair` to be removed. In the worst case, all n elements from the original vector might be added to `new_vector`.

- **Temporary Variables (O(1))**: There are constant size variables like loop iterators (`point`) and a flag for the conditional check that contribute negligible space complexity compared to the size of the new vector.

Overall Space Complexity:

The dominant space requirement comes from creating the new vector new_vector. Since it can hold up to n elements in the worst case, the space complexity of the function becomes $O(n)$. The detailed pseudo-code of the removePairFromVector algorithm is as follows:

```
removePairFromVector(point_vector, point_pair):
    new_vector ← []                                                          O(1)
    for point in point_vector:                                               O(n)
        if !((point.x == point_pair.first.x && point.y == point_pair.first.y) ||   O(1)
            (point.x == point_pair.second.x && point.y == point_pair.second.y)):   O(1)
            new_vector.push_back(point)                                      O(1)
    return new_vector                                                        O(1)
```

1.4. findClosestPairOrder Algorithm

1.4.1. Time Complexity Analysis

The time complexity of the findClosestPairOrder function depends on two main factors:

- **Loop Iterations (O(n))**:
  The while loop iterates at most n times (number of points) in the worst case.
- **Operations Inside the Loop**:
  - **closestPair Function (O(n log n))**:
    Assumed to have a time complexity of O(n log n) due to a divide-and-conquer approach for finding the closest pair in the current subset.
  - **removePairFromVector (O(n))**:
    The provided implementation iterates through the original vector in the worst case, leading to a time complexity of O(n) for removing elements.
  - **Other Operations (O(1))**:
    Conditional checks, swaps, and assignments inside the loop have constant time complexity (O(1)).

Overall Time Complexity:

There are two scenarios to consider:

- **Worst Case**:
  In the worst case, each loop iteration involves O(n log n) for closestPair and O(n)

for removePairFromVector. Therefore, the overall worst-case time complexity becomes:

$$O(n) \times (O(n \log n) + O(n)) = O(n^2 + n^2 \log n) = O(n^2 \log n)$$

- **Average Case (Hypothetical with Efficient removePairFromVector)**:
  Even though simplification is not possible in our current scenario, we can still consider a hypothetical case where removePairFromVector can be further optimized for constant time removal (e.g., using a hash table). This would hypothetically lead to an average-case time complexity of:

$$O(n) \times O(n \log n) = O(n^2 \log n)$$

Important Considerations:

- The actual time complexity might lie somewhere between the worst and average cases, depending on the efficiency of removing elements in removePairFromVector.
- It's crucial to analyze the specific implementation details of helper functions like removePairFromVector to get a more accurate understanding of the overall time complexity.

Overall Time Complexity:
The overall time complexity is as follows: in the worst case, it is $O(n^2 \log(n))$, and in the average case (hypothetical), it is also $O(n^2 \log(n))$, with simplification not possible in practice.

1.4.2. Space Complexity Analysis

The space complexity of the `findClosestPairOrder` function primarily depends on the following factors:

- **Input Points (O(n))**:
  The function needs to store the input points themselves, which takes O(n) space.
- **pairs Vector (O(n))**:
  The function uses a vector `pairs` to store the closest pairs found during the process. In the worst case, this vector can hold all n-1 closest pairs, leading to a space complexity of O(n).
- **Temporary Variables (O(1))**:
  There are constant size variables like `unconnected` and loop iterators that contribute negligible space complexity compared to the points and `pairs` vector.

Space Complexity:

Considering the dominant factors:

Worst Case: $O(n) + O(n) = O(2n)$

This occurs if all points form closest pairs and are stored in the `pairs` vector.

Important Considerations:

- In practice, the number of closest pairs might be less than n-1, leading to a space complexity closer to O(n).
- If additional data structures are used within `closestPair` with significant space requirements, the overall space complexity might be affected.

Overall Space Complexity:

The space complexity is $O(2n)$, indicating that the function utilizes a space proportional to twice the size of the input points. The detailed pseudo-code of the findClosestPairOrder algorithm is as follows:

```
findClosestPairOrder(points):
    pairs ← []                                                                    O(1)
    unconnected ← { -1, -1 }                                                      O(1)
    sort(points)                                                                  O(n log n)
    while points.size() > 1:                                                      O(n)
        closest_pair ← closestPair(points, 0, points.size())                      O(n log n)
        if compareY(closest_pair.second, closest_pair.first):                     O(1)
            swap(closest_pair.first, closest_pair.second)                         O(1)
        else if closest_pair.first.y == closest_pair.second.y && compareX(closest_pair.second, closest_pair.first):
                                                                                  O(1)
            swap(closest_pair.first, closest_pair.second)                         O(1)
        pairs.push_back(closest_pair)                                             O(1)
        points ← removePairFromVector(points, closest_pair)                      O(n)
    if !points.empty():                                                           O(1)
        unconnected ← points[0]                                                   O(1)
    for i from 0 to pairs.size():                                                 O(n)
        Print "Pair " + (i+1) + ": " + pairs[i].first.x + ", " + pairs[i].first.y + " - " + pairs[i].second.x + ", " +
pairs[i].second.y                                                                 O(1)
    if unconnected.x != -1:                                                       O(1)
        Print "Unconnected " + unconnected.x + ", " + unconnected.y               O(1)
```

1.5. Conclusion

The algorithms discussed in this section offer solutions to the closest pair of points problem using various approaches: Divide&Conquer, Brute-Force, and a hybrid method combining both. The Divide&Conquer algorithm achieves a time complexity of $O(n\log(n))$, making it efficient for large datasets, albeit with a space complexity of $O(n)$. Conversely, the Brute-Force method's time complexity is $O(n^2)$, leading to inefficiency for larger datasets. The hybrid approach in the findClosestPairOrder algorithm balances these trade-offs, combining $O(n\log(n))$ Divide&Conquer with $O(n^2)$ Brute-Force, resulting in an overall $O(n^2\log(n))$ time complexity. Ultimately, the choice of algorithm depends on factors like dataset size and performance requirements.

2. Performance Comparison of Divide&Conquer and Brute-Force Algorithms

2.1. Running Time Comparison

The table below is prepared to compare both algorithms' performances in different input data. The time taken is given in nanoseconds(ns).

	Case0	**Case1**	**Case2**	**Case3**	**Case4**
Divide&Conquer	23.273	27.501	92.133	345.807	1.193.602
Brute-Force	22.422	28.013	236.723	3.425.687	21.392.801

Table 2.1: Running Time Comparison of Divide&Conquer and Brute-Force on input data.

The provided table (Table 2.1) compares the running times of the divide-and-conquer and brute-force algorithms for finding the closest pair of points in different cases.

2.2. Time Complexity Analysis and Observations

Divide&Conquer: The divide-and-conquer approach has a time complexity of $O(n \log n)$, where n is the number of points. This means the running time grows logarithmically with the input size.

Brute-Force: The brute-force approach has a time complexity of $O(n^2)$. This signifies that the running time increases quadratically with the input size.

Observations from Table 2.1:

As expected from the theoretical complexities, the divide&conquer approach consistently outperforms the brute-force approach in all cases presented in Table 2.1. The difference in running time becomes more significant as the number of points (n) increases.

Here's a breakdown of the performance for each case:

1. **Case0**: In this case, the running time of the Divide & Conquer algorithm is slightly lower than that of the Brute-Force algorithm. Both algorithms seem to perform similarly, with negligible differences.

2. **Case1**: The running time for both algorithms increases slightly compared to Case0. However, the difference in performance between the two algorithms remains minimal.

3. **Case2**: Here, the running time for both algorithms increases significantly compared to the previous cases. The Divide&Conquer algorithm shows a slightly lower

running time compared to the Brute-Force algorithm, indicating its efficiency in handling larger input sizes.

4. **Case3**: The running time for both algorithms increases dramatically in this case, particularly for the Brute-Force algorithm. The Divide&Conquer algorithm continues to outperform the Brute-Force algorithm, showcasing its scalability and efficiency in handling larger input sizes.

5. **Case4**: This case represents the largest input size tested. The running time for both algorithms increases substantially, as expected. However, the Brute-Force algorithm's running time experiences a significant surge compared to the Divide&Conquer algorithm. This highlights the advantage of the Divide & Conquer algorithm in handling large datasets efficiently.

2.3. Why Divide&Conquer Performs Better:

The divide-and-conquer algorithm breaks down the problem into smaller subproblems, reducing the number of comparisons needed to find the closest pair. This results in a more efficient solution, especially for larger datasets. The logarithmic growth in running time with the input size makes it more scalable compared to the brute-force approach, whose running time explodes quadratically.

2.4. Conclusion:

For finding the closest pair of points, the Divide&Conquer approach is demonstrably superior in terms of time complexity and performance, especially for larger datasets. The Brute-Force approach might be suitable for very small datasets where the simplicity of implementation outweighs the potential performance penalty.

3. Effects of the Manhattan Distance on the Performance

3.1. Running Time Comparison and Observations

The table below is prepared to compare both algorithms' performances in different input data. The time taken is given in nanoseconds(ns).

	Case0	**Case1**	**Case2**	**Case3**	**Case4**
Divide&Conquer	13.886	15.539	61.705	230.100	773.486
Brute-Force	11.421	12.132	78.857	1.105.768	6.607.577

Table 3.1: Running Time Comparison of both algorithms on input data using Manhattan distance.

The provided tables (Table 3.1 and Table 2.1) showcase the running time comparison of the divide-and-conquer algorithm for finding the closest pair of points using both Manhattan and Euclidean distance metrics. The data is presented in nanoseconds (ns) for different input cases.

3.1.1. Manhattan vs. Euclidean Distance

While the divide-and-conquer algorithm remains the dominant approach in terms of efficiency, the choice of distance metric can subtly impact performance. In most cases, the tables suggest that Manhattan distance calculations are slightly faster than their Euclidean counterparts.

Potential Reasons:

This difference might be attributed to the computational simplicity of the Manhattan distance formula compared to the Euclidean distance, which involves a square root operation. Calculating the absolute difference between coordinates is generally less computationally expensive than calculating the square root of the sum of squared differences.

3.2. Results of Distance Method Used in findClosestPairOrder Algorithm

In this part, the resulting closest pair orders will be shown for both using the Euclidian Distance and Manhattan Distance. For simplicity some of the tables are formed with shortened output, normally Case2 has 50 pairs, Case3 has 200 pairs in addition to the 1 unconnected point, and Case4 has 500 pairs, only the first 10 pairs are given in the table for these cases.

Pair	First Pair	Second Pair
1	(9, 7)	(7, 8)
2	(14, 5)	(12, 9)
3	(17, 9)	(20, 14)
Unconnected	(1, 2)	

Table 3.2: Input Case0 and Euclidian Distance

Pair	First Pair	Second Pair
1	(9, 7)	(7, 8)
2	(12, 9)	(17, 9)
3	(14, 5)	(20, 14)
Unconnected	(1, 2)	

Table 3.3: Input Case0 and Manhattan Distance

In this case(Case0), the results are the same for both the Euclidian and Manhattan distance methods.

Pair	First Pair	Second Pair
1	(66, 34)	(57, 36)
2	(34, 42)	(29, 57)
3	(50, 70)	(48, 93)
4	(9, 12)	(51, 12)
5	(97, 30)	(1, 100)

Table 3.4: Input Case1 and Euclidean Distance

Pair	First Pair	Second Pair
1	(66, 34)	(57, 36)
2	(34, 42)	(29, 57)
3	(50, 70)	(48, 93)
4	(9, 12)	(51, 12)
5	(97, 30)	(1, 100)

Table 3.5: Input Case1 and Manhattan Distance

In this case(Case1), the results are the same for both the Euclidian and Manhattan distance methods.

Pair	First Pair	Second Pair
1	(656, 311)	(649, 316)
2	(555, 17)	(550, 26)
3	(850, 186)	(838, 192)
4	(838, 938)	(854, 946)
5	(931, 172)	(952, 177)
6	(216, 598)	(209, 621)
7	(800, 437)	(777, 450)
8	(738, 788)	(767, 788)
9	(33, 940)	(25, 968)
10	(825, 643)	(841, 670)

Table 3.6: Input Case2 and Euclidean Distance

Pair	First Pair	Second Pair
1	(656, 311)	(649, 316)
2	(555, 17)	(550, 26)
3	(850, 186)	(838, 192)
4	(838, 938)	(854, 946)
5	(931, 172)	(952, 177)
6	(738, 788)	(767, 788)
7	(216, 598)	(209, 621)
8	(800, 437)	(777, 450)
9	(33, 940)	(25, 968)
10	(825, 643)	(841, 670)

Table 3.7: Input Case2 and Manhattan Distance

In this case(Case2), the results are not the same for the Euclidian and Manhattan distance methods. As can be seen, the resulting pairs differ after pair 6, even if there are some partially the same pairs, overall the results are quite different from each other.

Pair	First Pair	Second Pair
1	(2677, 1058)	(2672, 1068)
2	(1572, 3490)	(1561, 3495)
3	(9275, 6793)	(9287, 6800)
4	(3392, 6272)	(3420, 6275)
5	(52, 2263)	(85, 2268)
6	(369, 1008)	(402, 1028)
7	(6369, 7184)	(6398, 7217)
8	(2184, 4160)	(2197, 4202)
9	(5321, 3451)	(5280, 3471)
10	(9672, 9071)	(9670, 9117)
Unconn	(2752, 9662)	

Table 3.8: Input Case3 and Euclidean Distance

Pair	First Pair	Second Pair
1	(2677, 1058)	(2672, 1068)
2	(1572, 3490)	(1561, 3495)
3	(9275, 6793)	(9287, 6800)
4	(3392, 6272)	(3420, 6275)
5	(52, 2263)	(85, 2268)
6	(9672, 9071)	(9670, 9117)
7	(4997, 5153)	(4994, 5202)
8	(369, 1008)	(402, 1028)
9	(2184, 4160)	(2197, 4202)
10	(5321, 3451)	(5280, 3471)
Unconn	(8152, 38)	

Table 3.9: Input Case3 and Manhattan Distance

In this case(Case3), the results are not the same for the Euclidian and Manhattan distance methods. As can be seen, the resulting pairs differ after pair 6 again, even if there are some partially the same pairs, overall the results are quite different from each other considering that the unconnected point is also different.

Pair	First Pair	Second Pair
1	(96529, 19949)	(96569, 19985)
2	(30737, 28343)	(30608, 28368)
3	(64970, 22389)	(64829, 22395)
4	(93652, 2621)	(93496, 2752)
5	(84449, 84409)	(84635, 84512)
6	(41918, 12087)	(41905, 12312)
7	(66554, 68961)	(66345, 69049)
8	(60939, 63742)	(60734, 63862)
9	(22539, 75606)	(22687, 75801)
10	(21767, 5437)	(21894, 5647)

Table 3.10: Input Case4 and Euclidean Distance

Pair	First Pair	Second Pair
1	(96529, 19949)	(96569, 19985)
2	(64970, 22389)	(64829, 22395)
3	(30737, 28343)	(30608, 28368)
4	(41918, 12087)	(41905, 12312)
5	(93652, 2621)	(93496, 2752)
6	(84449, 84409)	(84635, 84512)
7	(47221, 4637)	(47219, 4924)
8	(66554, 68961)	(66345, 69049)
9	(60939, 63742)	(60734, 63862)
10	(21767, 5437)	(21894, 5647)

Table 3.11: Input Case4 and Manhattan Distance

In this case(Case4), the results are not the same for the Euclidian and Manhattan distance methods. As can be seen, the resulting pairs differ after pair 2, even if there are some partially the same pairs, overall the results are significantly different from each other.

3.3. Reasons for the Difference in Results Using Different Distance Methods

The provided tables showcase a significant difference in the closest pair order identified for some cases (Case2, Case3, and Case4) when using Euclidean distance compared to Manhattan distance with the divide-and-conquer algorithm. This difference arises from the inherent nature of each distance metric:

- **Euclidean Distance:** This metric calculates the straight-line distance between two points, considering both the horizoental and vertical differences.
- **Manhattan Distance:** This metric calculates the total distance traveled along a grid, summing the absolute changes in x and y coordinates.

Impact on Closest Pair Identification:

For points clustered closely together, especially in a grid-like pattern, both metrics might identify similar closest pairs. This explains the initial matches observed in Case2, Case3, and Case4.

However, as the point distribution becomes more scattered or diagonal, the differences between the two metrics become more prominent. Euclidean distance prioritizes the straight-line distance, whereas Manhattan distance focuses on the grid-like movement. This can lead to them identifying different pairs as closest, as seen in the later results of the aforementioned cases.

Additional Considerations:

- The specific point distribution within each case plays a crucial role in how the distance metrics influence the closest pair identification.
- In some cases, the difference might be subtle, while in others, it can be significant, as seen in Case4 where the results diverge after only the second pair.

3.4. Conclusion

The discrepancies in the results obtained using different distance methods can be attributed to the fundamental differences between Euclidean and Manhattan distances. Euclidean distance measures the straight-line distance between two points in a two-dimensional space, while Manhattan distance calculates the distance by summing the absolute differences in the coordinates. Consequently, these distance metrics prioritize different geometric properties of the point sets. Euclidean distance emphasizes the diagonal distance, favoring pairs with relatively close proximity in both dimensions, whereas Manhattan distance emphasizes horizontal and vertical movements, favoring pairs with similar coordinates in one dimension but potentially differing significantly in the other. As a result, the ranking of pairs based on distance can vary significantly between the two methods, leading to divergent closest pair orders. Additionally, the choice of distance metric may also influence the sensitivity of the algorithm to outliers and noise in the dataset, further contributing to differences in the computed results. Hence, understanding the underlying geometric implications of each distance method is crucial for interpreting and comparing the outcomes accurately.

4. Conclusion

This study embarked on a detailed exploration of algorithms for finding the closest pair of points, with a specific focus on the context outlined in our file. We meticulously dissected both Divide&Conquer and Brute-Force approaches, analyzing their time and space complexities.

Through rigorous analysis and empirical testing, we demonstrated the superior performance of the Divide&Conquer method over Brute-Force, particularly as input sizes increase. This efficiency stems from the divide-and-conquer strategy, which significantly reduces computational overhead.

Additionally, we investigated the impact of distance metrics, particularly the Manhattan and Euclidean distances, on algorithmic performance. Our findings emphasized the importance of selecting the most suitable distance metric based on the problem domain, as it directly influences algorithmic accuracy and efficiency.

In concluding our study, we highlight the continuous quest for optimal solutions to computational problems. By leveraging the insights gained from this research, future endeavors in algorithmic development can aim to unravel even greater complexities and achieve heightened levels of efficiency and effectiveness.

Ultimately, this study contributes to the ongoing advancement of knowledge and understanding in algorithmic research, underscoring the significance of informed algorithm design choices in addressing real-world challenges.