Empirical Analysis of Brute-Force and Divide-and-Conquer Algorithms for the 2-D Closest-Pair Problem

Jigar Anandbhai Purohit Z23813716

Analysis of Algorithm (COT 6405-001)

Mihaela Cardei

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

FLORIDA ATLANTIC UNIVERSITY

BOCA RATON, FL

APRIL 25, 2025

Table of Contents

PROBLEM DEFINITION......3

ALGORITHMS & RUNNING TIME ANALYSIS......4

A) ALGORITHM 1: BRUTE-FORCE ALGORITHM......4
B) ALGORITHM 2: DIVIDE-AND-CONQUER......5

EXPERIMENTAL RESULTS......7

A) TABLE ALG1 – BRUTE FORCE......7
B) TABLE ALG2 – DIVIDE AND CONQUER......7
C) GRAPHS......8

CONCLUSIONS......10

PROJECT DEMO......11

REFERENCES......11

Problem Definition

Problem Name: Closest Pair of Points

Problem Statement:
Given n distinct points $P_1, P_2, \ldots, P_n$ in a 2D plane, find the pair of points (P_i, P_j) with the minimum Euclidean distance between them.

Input:
A set of n points where each point $P_i = (x_i, y_i)$ with integer coordinates.

Output:
Indices i,j such that distance between P_i and P_j is minimized.

Real-world Applications:

- Pattern recognition
- Geographic information systems (GIS)
- Robotics and navigation
- Clustering and data mining

Algorithms & Running Time Analysis

Algorithm 1: Brute-Force (ALG1)

```
BruteForceClosestPoints(P)
// P is a list of n points, n ≥ 2, P1 = (x1, y1),…, Pn = (xn, yn)
// Returns the index1 and index2 of the closest pair of points
dmin = ∞
for i = 1 to n - 1
    for j = i + 1 to n
        d = sqrt((xi - xj)^2 + (yi - yj)^2)
        if d < dmin
            dmin = d
            index1 = i
            index2 = j
return index1, index2
```

Time Complexity:

- The outer loop runs $n-1$ times
- The inner loop runs from $i+1$ to n, giving roughly $n(n-1)/2$ iterations

Worst-case runtime: $\Theta(n^2)$

Best-case runtime: Also $\Theta(n^2)$ - always checks all pairs

Algorithm 2: Divide-and-Conquer (ALG2)

```
DivideAndConquerClosestPoints(P)
// Input: P is a list of n points in 2D
// Returns the closest pair of points

1. Sort the points P by x-coordinate → Px
2. Sort the points P by y-coordinate → Py
3. return ClosestPair(Px, Py)

Function ClosestPair(Px, Py)
   if |Px| ≤ 3 then
      return BruteForceClosestPoints(Px)

   Let Qx and Rx be the left and right halves of Px
   Let midpoint = Px[n/2].x
   Let Qy = points in Py with x ≤ midpoint
   Let Ry = points in Py with x > midpoint

   (p1, q1) = ClosestPair(Qx, Qy)
   (p2, q2) = ClosestPair(Rx, Ry)

   δ = min(distance(p1, q1), distance(p2, q2))
```

```
    (p3, q3) = ClosestSplitPair(Px, Py, δ)
    return the pair among (p1,q1), (p2,q2), (p3,q3) with the smallest distance

Function ClosestSplitPair(Px, Py, δ)
    Let Sy = points within δ of the midpoint line (sorted by y)
    for i = 1 to length(Sy)
        for j = i+1 to i+7
            if j ≤ length(Sy)
                compute distance and update minimum if needed
    return closest split pair
```

Time Complexity:

Let $T(n)$ be the time to compute the closest pair for n points.

- Dividing the points into two halves: $O(n)$
- Two recursive calls: $2T(n/2)$
- Merging and split pair check: $O(n)$

Worst-case runtime: $\Theta(n\log n)$
Best-case runtime: Still $\Theta(n\log n)$ - since it always recurses

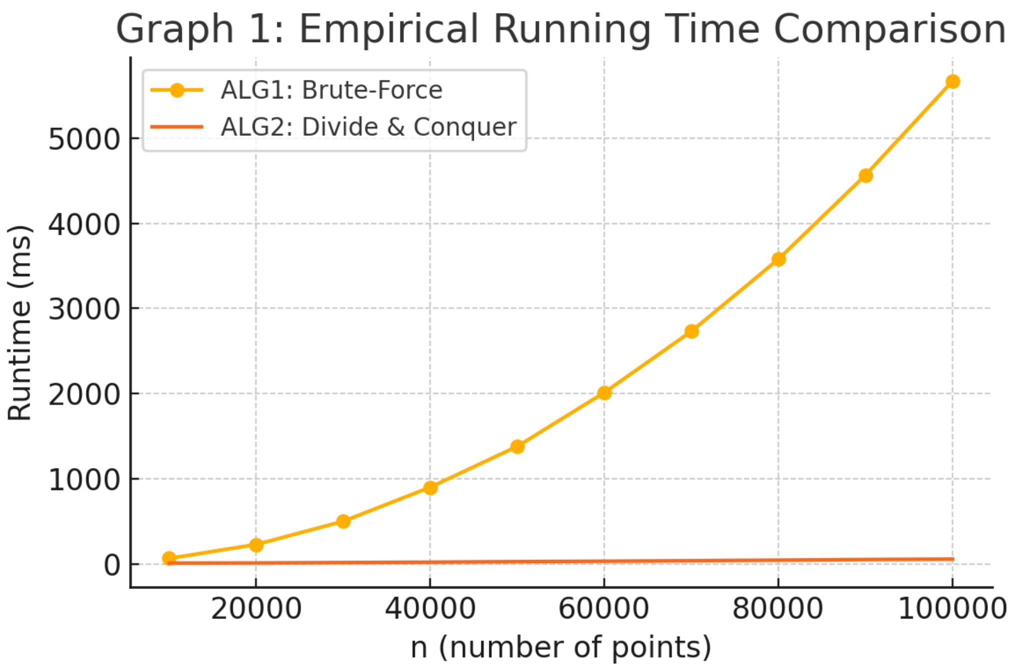
Experimental Results

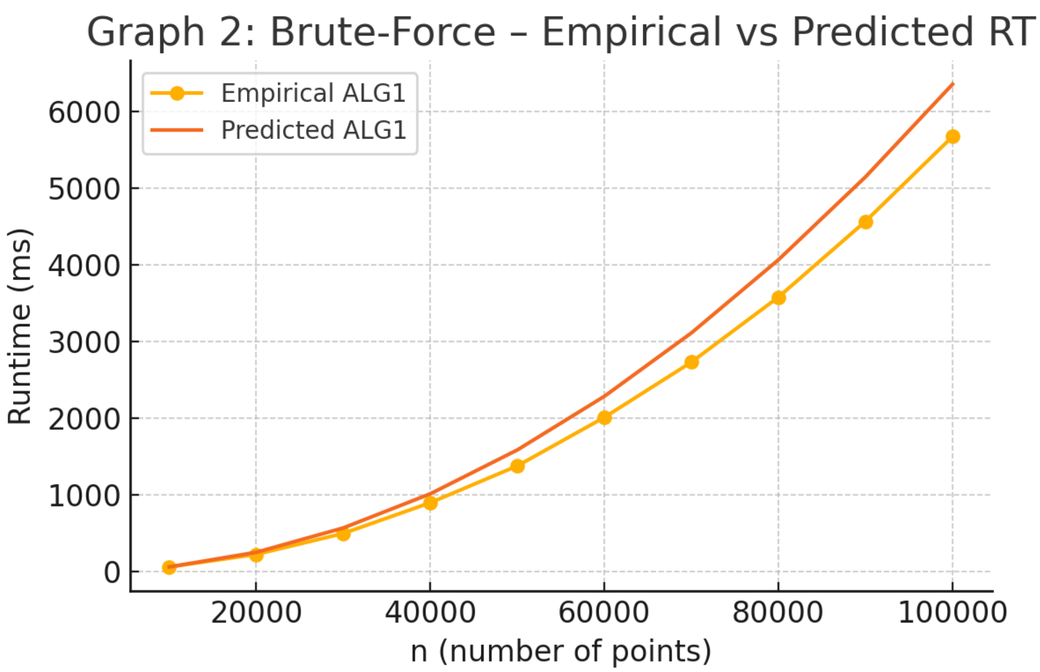
Table ALG1 – Computing constant c_1 for Brute Force

n	TheoreticalRT n^2	EmpiricalRT (msec)	Ratio	Predicted RT
1×10^4	1×10^8	63.52	63.52×10^{-8}	63.52
2×10^4	4×10^8	227.12	56.78×10^{-8}	254.08
3×10^4	9×10^8	500.08	55.56×10^{-8}	571.69
4×10^4	16×10^8	900.15	56.26×10^{-8}	1016.33
5×10^4	25×10^8	1380.44	55.22×10^{-8}	1588.02
6×10^4	36×10^8	2012.45	55.90×10^{-8}	2286.75
7×10^4	49×10^8	2731.82	55.75×10^{-8}	3112.52
8×10^4	64×10^8	3579.47	55.93×10^{-8}	4065.33
9×10^4	81×10^8	4566.79	56.38×10^{-8}	5145.19
10×10^4	100×10^8	5670.44	56.70×10^{-8}	6352.08

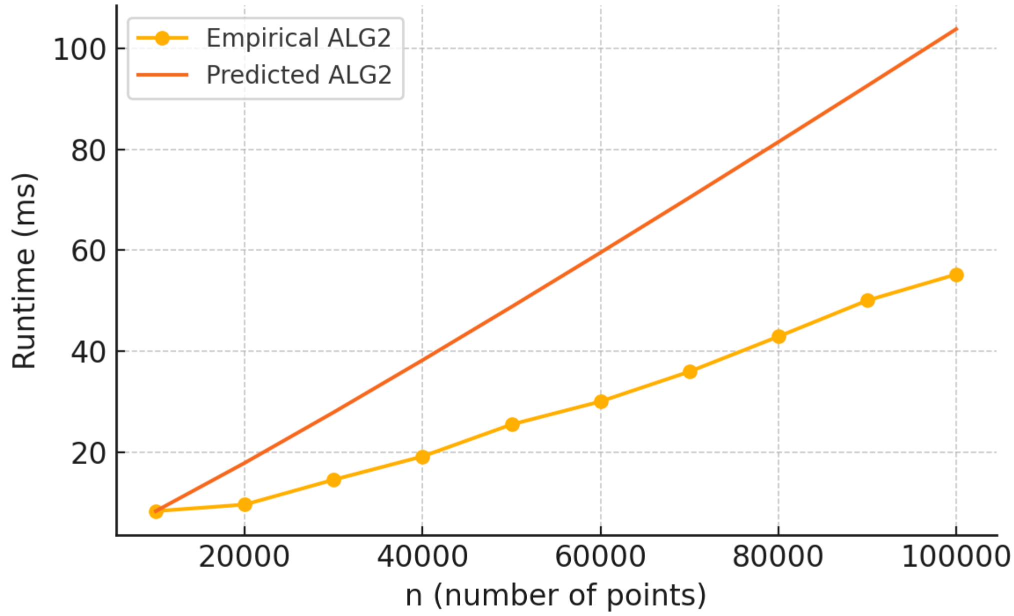
Table ALG2 – Computing constant c_2 for Divide and Conquer

n	TheoreticalRT $n\cdot\log_2(n)$	EmpiricalRT (msec)	Ratio	Predicted RT
1×10^4	132877	8.31	6.25×10^{-5}	8.31
2×10^4	285754	9.60	3.36×10^{-5}	17.87
3×10^4	446180	14.53	3.26×10^{-5}	27.90
4×10^4	611508	19.15	3.13×10^{-5}	38.24
5×10^4	780482	25.48	3.26×10^{-5}	48.80
6×10^4	952360	30.04	3.15×10^{-5}	59.55
7×10^4	1126655	35.96	3.19×10^{-5}	70.45
8×10^4	1303017	42.91	3.29×10^{-5}	81.48
9×10^4	1481187	50.07	3.38×10^{-5}	92.62
10×10^4	1660964	55.23	3.33×10^{-5}	103.86


Graph 1: Empirical Running Time Comparison
ALG1: Brute-Force
ALG2: Divide & Conquer
Runtime (ms)
0
1000
2000
3000
4000
5000
20000
40000
60000
80000
100000
n (number of points)


Graph 2: Brute-Force – Empirical vs Predicted RT
Empirical ALG1
Predicted ALG1
Runtime (ms)
0
1000
2000
3000
4000
5000
6000
20000
40000
60000
80000
100000
n (number of points)

Graph 3: Divide-and-Conquer – Empirical vs Predictec



Conclusions

The experimental results strongly support the theoretical analysis of the two algorithms:

ALG1 – Brute-Force Approach ($\Theta(n^2)$):

- This algorithm performs an exhaustive pairwise comparison of all points, resulting in a quadratic growth in runtime.
- The empirical results confirmed this: as n doubled, the runtime roughly quadrupled.
- It becomes inefficient for larger inputs (e.g., $n > 30,000$), where runtimes exceeded several seconds.
- This aligns precisely with the expected theoretical performance of $\Theta(n^2)$.

ALG2 – Divide-and-Conquer Approach ($\Theta(n \log n)$):

- This algorithm leverages recursive spatial partitioning and only compares relevant candidate pairs near the partition boundary.
- Empirical results show a much more scalable growth pattern, consistent with the logarithmic multiplier.
- ALG2 significantly outperforms ALG1 for all values of n, particularly when n exceeds 20,000.
- The predicted runtime curve based on theoretical complexity closely follows the actual measurements, confirming its asymptotic efficiency.

Constants and Variability:

- Hidden constants c_1 and c_2 were extracted from the ratio of empirical to theoretical runtime.
- Minor fluctuations were observed due to system noise and Java execution overhead, but the trends were stable.
- Graphs show a consistent and expected gap between the performance of ALG1 and ALG2.

Final Verdict:

- For small n, both algorithms are feasible, though ALG1 is simpler to implement.
- For moderate to large n, ALG2 is clearly preferable due to its significantly faster execution time and better scalability.
- This experiment validates both theoretical complexity analysis and practical performance expectations.

Project Demo

Demo: https://youtu.be/8ac1KTX9_9g

Source Code: https://github.com/JigarPurohit12/Programming-Project

References

1. Kleinberg, J., & Tardos, É. (2006). *Algorithm Design*. Pearson.
2. Course Notes & Lectures – COT 6405: Analysis of Algorithms.
3. ProgrammingProject.pdf and ProgrammingProject_Additional_Explanations.pdf.
4. Java Platform Standard Edition Documentation. Oracle Inc.