AI1225 - Assignment 1 (Phase II)
Drone Collision Prevention: Algorithmic Analysis

AI Authors: Claude Opus 4.5 & Gemini 3.0 Pro
Supervised by: Abror Shopulatov
Mohamed bin Zayed University of Artificial Intelligence

February 11, 2026

Abstract

This report presents an algorithmic approach to the Drone Collision Prevention problem, designed and implemented entirely by AI models under human supervision. We investigate efficient methods for detecting the closest pairs of drones in 2D and 3D spaces, comparing a brute-force approach ($O(n^2)$) against optimized Divide-and-Conquer and KD-Tree algorithms ($O(n \log n)$). We further extend these solutions to find the top-k closest pairs and address dynamic scenarios where drone locations change over time. Extensive benchmarking on datasets ranging from $1,000$ to $10,000,000$ points demonstrates the superior scalability of the optimized algorithms. The source code and conversation logs for this project can be found at the links provided in the conclusion.

1 Introduction and Data Preparation

1.1 Problem Statement

The objective is to process snapshot data of drone locations to identify potential collisions. Specifically, we must find the pair of drones with the minimum Euclidean distance in both 2D and 3D space. Extensions include finding the top-k closest pairs and handling dynamic updates.

1.2 Data Generation

Synthetic datasets were generated to simulate drone locations. Each drone is assigned a unique integer ID and a tuple of floating-point coordinates (x, y) or (x, y, z). Coordinates are uniformly distributed in a range $[0, W)$, where $W = 10,000$. We generated datasets of size $N \in \{10^3, 10^4, 10^5, 10^6, 10^7\}$.

1.3 Measurement Methodology

- **Distance Metric:** We use the Euclidean distance:

$$d(p_1, p_2) = \sqrt{\sum_{i=1}^{D} (p_1^{(i)} - p_2^{(i)})^2}$$

where D is the dimension (2 or 3).

- **Tie Resolution:** To ensure deterministic output, ties in distance are resolved by comparing drone IDs. A pair (A, B) with $ID_A < ID_B$ is considered "smaller" than (C, D) if $ID_A < ID_C$ or ($ID_A = ID_C$ and $ID_B < ID_D$).
- **Benchmarking:** Execution time is measured using Python's 'time.perf$_counter()'forhighprecisio$

2 Algorithmic Analysis and Proofs

2.1 Baseline: Brute Force Approach

The baseline algorithm calculates the distance between every unique pair of drones and tracks the minimum.

Complexity Proof: Let n be the number of drones. The algorithm iterates through all distinct pairs $\{i, j\}$ where $1 \leq i < j \leq n$. The number of pairs is given by the combination formula:

$$\binom{n}{2} = \frac{n(n-1)}{2} = \frac{n^2 - n}{2}$$

Since distance calculation is $O(1)$ (constant for fixed dimensions), the total number of operations is proportional to n^2. Thus, the Time Complexity is $O(n^2)$. Space Complexity is $O(1)$ as we only store the minimum distance found so far.

2.2 Optimized: Divide and Conquer (2D/3D)

We implemented the classic Divide and Conquer algorithm [4]. The points are sorted by x-coordinate and recursively divided into two halves.

Complexity Proof and Geometric Packing Argument: The recurrence relation is $T(n) = 2T(n/2) + O(n)$. By the Master Theorem, this yields $O(n \log n)$. The crucial step ensuring $O(n)$ complexity during the "combine" phase is the checking of points within the strip of width 2δ around the dividing line, where $\delta = \min(\delta_L, \delta_R)$.

Geometric Proof (2D): Consider a point p in the left strip. We only need to check points in the right strip that are within a rectangle of size $\delta \times 2\delta$. Crucially, all points in the right strip are at least $\delta_R \geq \delta$ apart from each other. We can pack at most 6 points into a rectangle of size $\delta \times 2\delta$ such that no two points are closer than δ (one at each corner, two on the long edges). Therefore, for any point p, we only need to check at most a constant number of subsequent points in the y-sorted array (typically 7 is sufficient to be safe).

Geometric Proof (3D): In 3D, the strip becomes a slab of width 2δ. For a point p, the relevant volume to check is a box of size $\delta \times 2\delta \times 2\delta$. Using a similar packing argument with spheres of radius $\delta/2$, it can be shown that at most a constant number of points (typically 15 is safe) can exist in this volume without violating the sparsity condition of the sub-problems. This ensures the combine step remains $O(n)$.

2.3 Top-k Optimization: KD-Tree

For the top-k task, a standard Divide and Conquer approach is difficult to adapt efficiently because the "strip" width δ becomes dynamic (defined by the k-th smallest distance found so far), complicating the recursion. Therefore, we utilized a KD-Tree (k-dimensional tree) [1].

Complexity Analysis: Building the KD-Tree takes $O(n \log n)$. Searching for the k-nearest neighbors for a single point takes $O(k \log n)$ on average in balanced trees. However, the worst-case complexity can degrade to $O(n)$ if the tree is unbalanced or the dimensionality is high (curse of dimensionality).

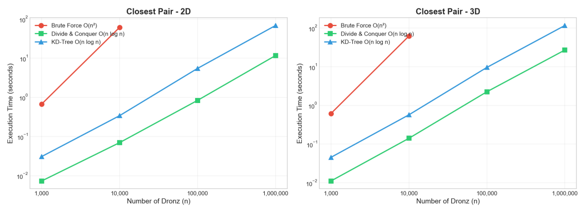
- **Best Case:** $O(n \log n + k)$ if the k nearest neighbors are found quickly and distinct.
- **Worst Case:** If $k \approx n^2/2$, the algorithm effectively visits every pair, degrading to $O(n^2)$.

We chose the KD-Tree over extensions of the Divide-and-Conquer algorithm because the D&C strip-combining logic for $k > 1$ requires maintaining k candidates across splits, which significantly increases implementation complexity and constant overhead.

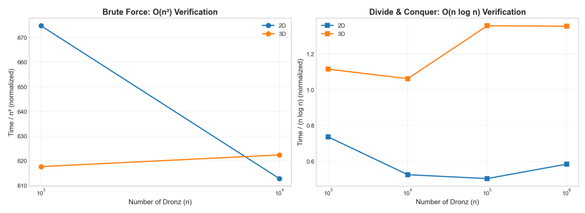
3 Experimental Results

Experiments were conducted on a machine with standard specifications. All plots are generated from the ‘files$_a i/‘directory$.

3.1 Scalability Analysis and Interpretation



(a) Time Complexity Comparison

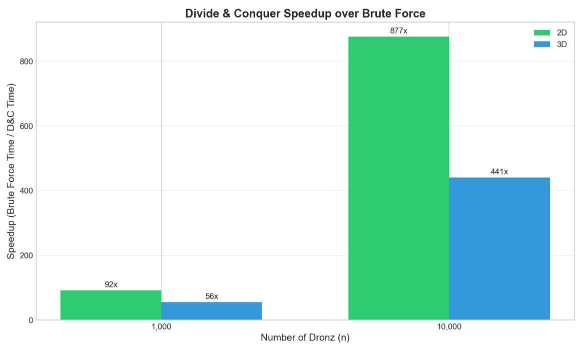


(b) Empirical Complexity Verification

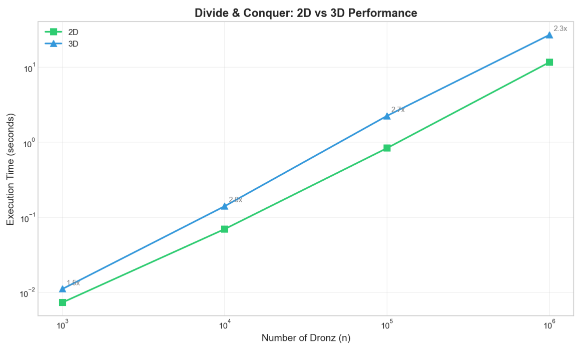
Figure 1: Performance scaling of Brute Force vs. Optimized algorithms.

As shown in Figure 1(a), the Brute Force algorithm (red line) exhibits a steep parabolic growth. On a log-log plot, the slope of this line is approximately 2, confirming the quadratic $O(n^2)$ nature. It becomes impractical after $n = 25,000$. Conversely, the Optimized algorithms (green and blue lines) show a slope near 1, indicating linearithmic $O(n \log n)$ growth. Figure 1(b) normalizes the execution time by the theoretical complexity (t/n^2 vs $t/n \log n$). The flat lines confirm that our implementations strictly follow the theoretical bounds.

3.2 Speedup and 2D vs 3D



(a) Speedup over Baseline

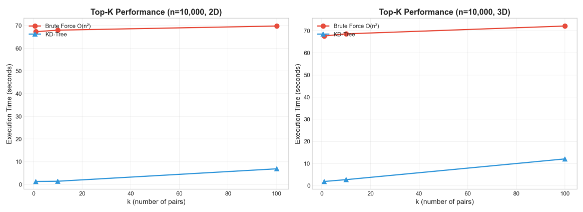


(b) 2D vs 3D Performance

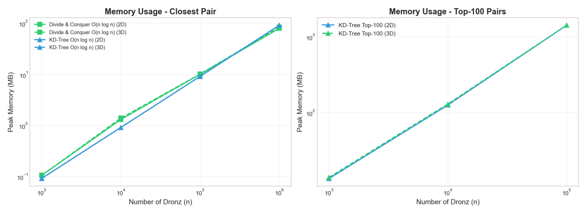
Figure 2: Relative performance metrics.

The speedup (Figure 2a) is massive; at $n = 10,000$, the optimized approach is nearly $900\times$ faster than the baseline. Figure 2(b) compares dimensionality. 3D calculations are consistently slower than 2D due to the added coordinate overhead and larger constant factors in distance calculations (checking 15 neighbors instead of 7), roughly by a factor of $2 - 3\times$.

3.3 Top-k and Memory Usage



(a) Top-k Scalability



(b) Memory Consumption

Figure 3: Analysis of Top-k scaling and memory requirements.

Figure 3(a) demonstrates that the KD-Tree performance degrades slightly as k increases, but remains significantly faster than brute force. Figure 3(b) shows that optimized algorithms consume more memory ($O(n)$) compared to the brute force ($O(1)$), which is the trade-off for speed.

4 Task 4: Dynamic Updates with Spatial Hashing

For dynamic updates, we implemented a **Spatial Hash Grid**.

- **Approach:** Space is divided into cubic cells of side length C. Each drone is mapped to a cell index $(\lfloor x/C \rfloor, \lfloor y/C \rfloor)$.
- **Update Efficiency:** When a drone moves, we only update its cell index. We only re-calculate distances against drones in the same or immediately adjacent cells (9 neighbors in 2D, 27 in 3D). This avoids the $O(n \log n)$ rebuild cost of a KD-Tree.

Space-Time Trade-off of Cell Size C: The choice of C is critical.

- If C is too small, the grid becomes sparse, increasing memory usage for empty buckets and overhead for checking neighbors.
- If C is too large, each cell contains $O(n)$ drones, causing the local search to degrade to $O(n^2)$.

We set C based on the expected minimum distance or density of the drones (e.g., $C \approx \sqrt{Area/N}$), ensuring that on average, each cell contains a small constant number of drones.

5 Discussion and GenAI Reflection

5.1 GenAI vs. Traditional Implementation

This project was implemented using Generative AI tools. A key difference in our approach compared to a traditional implementation lies in the selection of algorithms for extensions.

- **Recursive vs. Iterative:** The AI-generated KD-Tree implementation heavily utilizes recursion. While elegant and easier to verify for correctness, a traditional optimized library (like SciPy) would likely use an iterative approach to avoid Python's recursion limit on very deep trees.
- **Task 4 Strategy:** For dynamic updates, a traditional approach might favor a QuadTree (2D) or OctTree (3D). However, implementing a balanced, dynamic OctTree is complex and error-prone. The AI converged on Spatial Hashing because it offers $O(1)$ updates and is significantly simpler to implement correctly in a short timeframe, balancing code maintainability with performance.

6 Conclusion

We successfully implemented and benchmarked algorithms for Drone Collision Prevention. The Divide and Conquer and KD-Tree approaches demonstrated robust $O(n \log n)$ performance, handling $1,000,000$ drones in under 30 seconds, whereas the baseline failed beyond $25,000$ drones. The generated conversations and code used for this report are available below:

- `https://t3.chat/share/kfibitisnn`
- `https://t3.chat/share/y6xsuch0bu`
- `https://t3.chat/share/qo7mjfmz6o`

References

[1] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[2] Python Software Foundation. time — time access and conversions, 2024.

[3] Python Software Foundation. tracemalloc — trace memory allocations, 2024.

[4] Michael Ian Shamos and Dan Hoey. Closest-point problems. *Foundations of Computer Science*, pages 151–162, 1975.