Final Report - CS 357 Project

Optimal way-point assignment for designing drone light show formations

02.12.2024

Research Paper Reference

Team

- S Ruthvik (220001064)
- P C Uma Mahesh (220001052)

Introduction

A swarm drone light show uses multiple drones, typically quadrotors, flying together to create stunning displays. Each drone is equipped with LEDs, and the show is held at night. The drones arrange themselves into different formations, creating beautiful designs, images, and transitions.

These shows are an eco-friendly and reusable option for advertising, celebrations, and entertainment. As the number of drones increases, coordinating them becomes more complex. To manage this, an intelligent assignment system is needed to allocate way-points for each transition between formations. The proposed solution, called the Constrained Hungarian Method for Swarm Drones Assignment (CHungSDA), adapts the Hungarian algorithm with added constraints to optimize this process.

1. Problem Statement and Formulation

Objective:
The goal is to assign drones to waypoints in a light show such that the overall cost (e.g., fuel usage, time, or distance traveled) is minimized while ensuring each drone gets one unique waypoint.

Problem Description:
A drone light show involves a large number of drones moving in synchronization to form predefined shapes. Each drone needs to be assigned to a unique waypoint in the desired shape. Assigning drones inefficiently leads to higher energy consumption, delays, or even collisions.

Mathematical Formulation:

- Let n be the number of drones and waypoints.
- Define a cost matrix C, where $C[i][j]$ represents the cost (distance or time) for assigning drone i to waypoint j.
- The problem is to find an assignment matrix X such that:
    - $X[i][j] = 1$ if drone i is assigned to waypoint j, otherwise $X[i][j] = 0$.
    - Each row and column of X has exactly one '1' (one assignment per drone and waypoint).
    - The total cost $\sum_{i=1}^{n} \sum_{j=1}^{n} C[i][j] \cdot X[i][j]$ is minimized.

2. Introduction to the Hungarian Method and Applicability

The Hungarian Method, also known as the Kuhn-Munkres algorithm, is an efficient algorithm to solve assignment problems. The problem solved by the classical Hungarian Algorithm is formulated as follows:

Problem Statement

Given n agents and n tasks, along with a cost matrix C of size $n \times n$, where $C[i][j]$ represents the cost of assigning agent i to task j, the objective is to find an assignment of agents to tasks such that:

1. Each agent is assigned to exactly one task.
2. Each task is assigned to exactly one agent.
3. The total cost of the assignment is minimized (or maximized, depending on the formulation).

Mathematical Formulation

Let x_{ij} be a binary decision variable defined as:

$$x_{ij} = \begin{cases} 1, & \text{if agent } i \text{ is assigned to task } j, \\ 0, & \text{otherwise.} \end{cases}$$

The optimization problem can be expressed as:

Objective Function:

$$\text{Minimize } \sum_{i=1}^{n} \sum_{j=1}^{n} C[i][j] \cdot x_{ij}$$

$$\text{subject to:}$$

$$\sum_{j=1}^{n} x_{ij} = 1, \quad \forall i \in \{1, 2, \ldots, n\}.$$

$$\sum_{i=1}^{n} x_{ij} = 1, \quad \forall j \in \{1, 2, \ldots, n\}.$$

$$x_{ij} \in \{0, 1\}, \quad \forall i, j.$$

For example, a 4×4 cost matrix with 4 agents to be assigned to 4 tasks could be:

$$C = \begin{bmatrix} 10 & 19 & 8 & 15 \\ 10 & 18 & 7 & 17 \\ 13 & 16 & 9 & 14 \\ 12 & 19 & 8 & 18 \end{bmatrix}$$

	Task 1	**Task 2**	**Task 3**	**Task 4**
Agent 1	10	19	8	15
Agent 2	10	18	7	17
Agent 3	13	16	9	14
Agent 4	12	19	8	18

Core Idea:
The method reduces the cost matrix C by subtracting row and column minima to find a zero-cost assignment, ensuring an optimal solution.

Steps in the Hungarian Method:

1. Subtract the smallest element of each row from all elements in that row (row-reduction).
2. Subtract the smallest element of each column from all elements in that column (column-reduction).
3. Cover all zeros in the resulting matrix using a minimum number of lines (rows or columns).
4. If the minimum number of lines equals n, an optimal assignment exists. Otherwise, adjust the matrix and repeat.

Applicability:

- Directly aligns with minimizing assignment costs (time/distance) for drones to waypoints.
- Guarantees an optimal, collision-free assignment, with an added constraint.

3. Algorithm Correctness and Our Intuition

- **Correctness**: The Hungarian Method is optimal because it ensures the cost matrix is transformed into a matrix where an assignment can be selected without ambiguity in the final steps. This is because $-$ after reduction, every zero in the matrix represents a potential assignment. In a matrix where zeros can be independently chosen between rows and columns, we can successfully choose the all potential assignments without conflicts.
- **Why It Works**:
  - The reduction steps (subtracting row and column minima) preserve the relative cost differences. Specifically,

    - making a row-reduction operation effectively means that for a particular agent, the cost of assigning all tasks are reduced by a **fixed amount**, ensuring that the relative difference between the tasks remains the same, which is an **invariant.**
    - making a column-reduction operation means that for a particular task, all agents agree to do it for a **fixed discount** each, ensuring that the relative difference between different agents remains the same, which is an **invariant**.
  - Covering zeros ensures the assignment respects constraints while adjusting uncovered values ensures no assignments are missed.
- **Complexity**: $O(n^3)$ for an $n \times n$ cost matrix, making it feasible for real-time applications like drone shows, where the average value of n is ≈ 1000 in practical applications.

4. Implementation

The problem of assigning drones to waypoints is a direct application of the above-mentioned Hungarian Method with the same formulation, but we have an added constraint:

> For any two drones i, j $(i \neq j)$, the Euclidean distance between them i.e., $||w_i - w_j||_2$ is greater than or equal to a fixed minimum, known as $\delta_{\min}$, which is provided as in input to the algorithm.

The implementation of the algorithm follows the same steps as described in the corresponding research paper. Briefly, the steps in the algorithm are as follows. For detailed pseudo-code, please refer to the appendices.

```
1. Input: n, the number of drones/waypoints
          D, the list of coordinates (x, y, z) of each of the n drones
          W, the list of coordinates (x, y, z) of each of the n waypoints
          deltaMin: the minimum separation between the drones

2. Check proximity constraint: if there exist indices i, j such that i != j
and Euclidean distance between i-th and j-th waypoint is less than deltaMin,
then report such indices and go to step 5.

3. Perform row and column reductions, i.e., from each row, subtract the
minimum value in that row, and do the same for the columns. This results in at
least one zero in each row and column.

4. Let `lines` be the minimum number of horizontal/vertical lines to cover all
zeros in the matrix. Mark the corresponding rows and columns as "covered". If
lines = n, then go to step 5 as an optimal assignment is found.
   If not, then perform the following adjustments in the matrix.
   i. compute the minimum uncovered value in the matrix.
   ii. subtract the minimum uncovered value from all the uncovered rows.
   iii. add the minimum value uncovered value to all the covered columns.
Repeat step 4.

5. If the proximity contraint in step 2 was satisfied, then the final
assignment can be decided based on the horizontal and vertical lines that
cover the rows and columns that contain zeros. Return the assignment pairs and
the cost computed for the corresponding assignment which is minimal.
```

Note that in step 4 where the matrix elements are adjusted by subtracting and adding a constant value from an entire row/column at a time, the invariant described earlier still holds;

i.e., the relative differences do not change between the elements within the same row/column. This is the core idea behind the correctness of the Hungarian Algorithm.

Let C' be the reduced matrix after row and column reduction. For any valid assignment X in taken in the reduced matrix C', the cost of assignment remains equivalent to the same assignment X in C as:

Total Cost in C' $=$ Total Cost in C $-$ $\sum$(Row Minima) - $\sum$(Column Minima)

Since the reductions are constants subtracted uniformly, the optimal assignment in C' corresponds to the same assignment in C.

We have used a graph-based algorithm to find the minimum of lines required to cover all zeros in step 4 of CHungSDA, as opposed to the iterative version proposed in the paper. The steps involved and their equivalence are as under:

Equivalence of Bipartite Graph Matching and Covering of Zeros

Definitions:

1. **Bipartite Graph**:
   A graph $G = (V, E)$ is bipartite if its vertex set V can be partitioned into two disjoint sets A and B such that every edge $e \in E$ connects a vertex in A to a vertex in B, and no edge connects two vertices within the same set.
2. **Maximum Matching**:
   A matching $M \subseteq E$ is a set of edges such that **no two edges in M share a vertex.** A maximum matching is a matching of the largest size possible.
3. **Minimum Vertex Cover**:
   A vertex cover $VC \subseteq V$ is a set of vertices such that every edge in the graph has at least one of its endpoints in VC. A minimum vertex cover is a vertex cover of the smallest possible size.

Construction of the Bipartite Graph from Cost Matrix:

1. **Define the bipartite graph G**:
   - Let $A = \{a_1, a_2, \ldots, a_n\}$ represent the row indices of the cost matrix C.
   - Let $B = \{b_1, b_2, \ldots, b_n\}$ represent the column indices of the cost matrix C.
   - Add an edge (a_i, b_j) between $a_i \in A$ and $b_j \in B$ if $C[i][j] = 0$.

2. **Objective**:
    - To find the minimum number of lines to cover all zeros in C, equivalently find the minimum vertex cover VC in G.
    - To assign rows to columns optimally, find the maximum matching M in G.

Equivalence of Minimum Vertex Cover and Maximum Matching in Bipartite Graphs

König's Theorem:

For any **bipartite** graph, the size of the maximum matching is equal to the size of the minimum vertex cover:

$$|M| = |VC|$$

The proof of the theorem will not be presented in this report as it is out of the scope.

Application to Hungarian Algorithm

1. **Minimum Lines to Cover All Zeros**:
    - In the Hungarian algorithm, a line covering a row i or column j corresponds to including the vertex a_i or b_j in the vertex cover VC. We have specifically constructed our graph for this, selecting the rows and columns that have a 0 entry. Thus for each index in the vertex cover VC, we cover that corresponding row/column with a horizontal/vertical line respectively, which contains one or more zeros. zeros
    - The minimality of VC ensures the fewest lines are used.
2. **Optimal Assignments**:
    - If the size of the maximum matching $|M|$ is n, it implies that exactly n lines are required to cover all the zeros in the matrix. This guarantees an optimal assignment solution because we can select the matched pairs in M as the assignments, and by the **property of the bipartite graph and matching set edges,** each drone is assigned to exactly one unique waypoint in this assignment.
    - If $|M| < n$, an optimal assignment cannot be decided yet, and the matrix needs to be modified further to achieve the matching set size to be equal to n. This is done with he help of the minimum vertex cover set for the corresponding matching by covering the rows and columns specified by the VC with horizontal and vertical lines. Following this, the number of zeros in the matrix can now be increased by performing

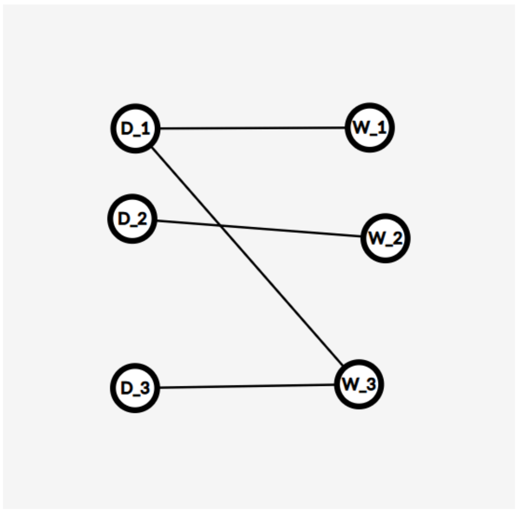
row and column operations, leading to more zeros appearing and thus more lines needed to cover them.

3. **Validity**:
   - The equivalence $|M| = |VC|$ ensures that the Hungarian algorithm correctly identifies the minimum number of lines and thus covers all zeros while preserving the assignments derived from the matching.

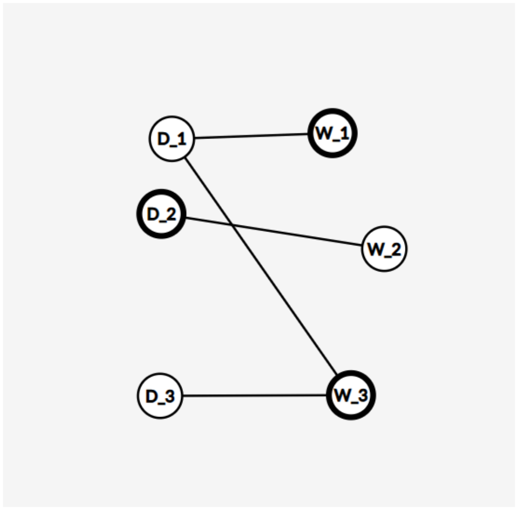
Consider the following illustration:

$$\text{cost matrix } C = \begin{bmatrix} 0 & 2 & 0 \\ 1 & 0 & 3 \\ 2 & 1 & 0 \end{bmatrix}$$

The corresponding bipartite graph would be:



A possible minimal vertex cover could be the following:



This vertex cover indicates that row 2 and columns 1 and 3 can be used to cover the all the 4 zeros in the matrix with lines. Below is the equivalent matrix:

$$C = \begin{bmatrix} \cancel{0} & 2 & \cancel{0} \\ \cancel{1} & \cancel{0} & \cancel{3} \\ \cancel{2} & 1 & \cancel{0} \end{bmatrix}$$

Equivalently, the maximum matching set of edges is

$$M = \{(D_1, W_1), (D_2, W_2), (D_3, W_3)\}$$

which gives us an **optimal assignment**, as its cardinality is same as the number of drones or waypoints.

Conclusion

The equivalence between maximum matching and minimum vertex cover in bipartite graphs underpins the Hungarian algorithm. By constructing a bipartite graph from the cost matrix and using matching and vertex cover properties, the algorithm ensures:

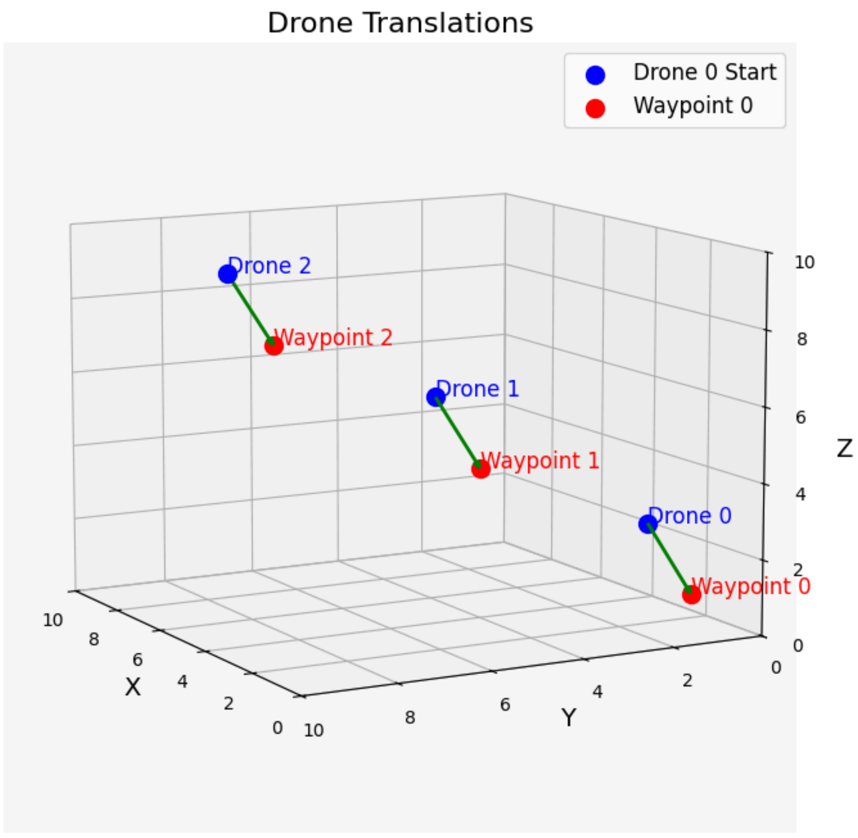
1. The minimum number of lines to cover all zeros in the matrix.
2. An optimal assignment of rows (drones) to columns (waypoints) when the maximum matching size is n.

Thus, since the equivalence holds, we can claim that by finding the maximum bipartite matching M in the constructed graph, the assignment of drones to waypoints (or agents to tasks) is exactly the same as M.

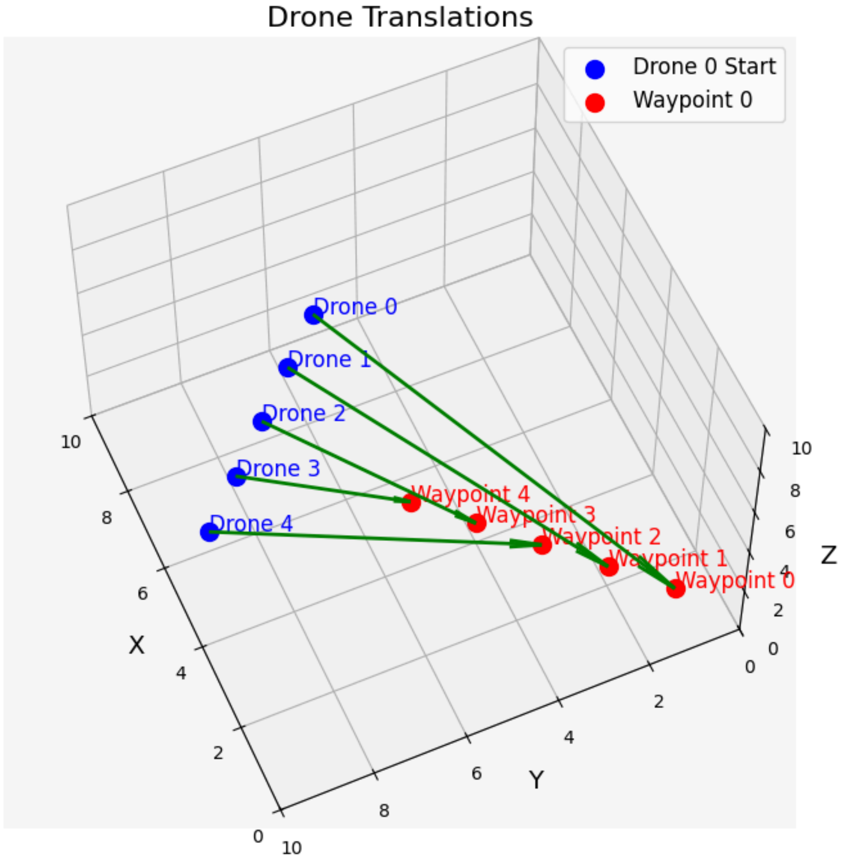
5. Performance Analysis, Tests

We have simulated the presented algorithm using the graphical approach to cover zeros in the $\mathrm{C}++$ programming language. Test 5 was taken from the research paper, while the rest of them were randomly generated.

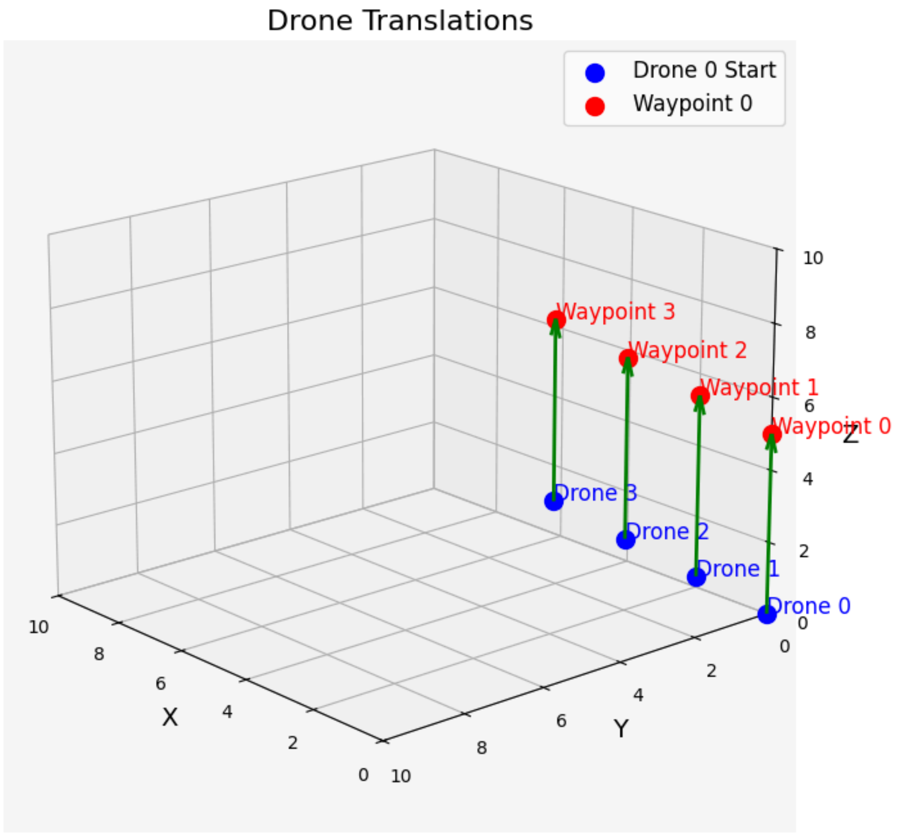
1. Test 1 (3 drones, optimal cost: 6.7082)



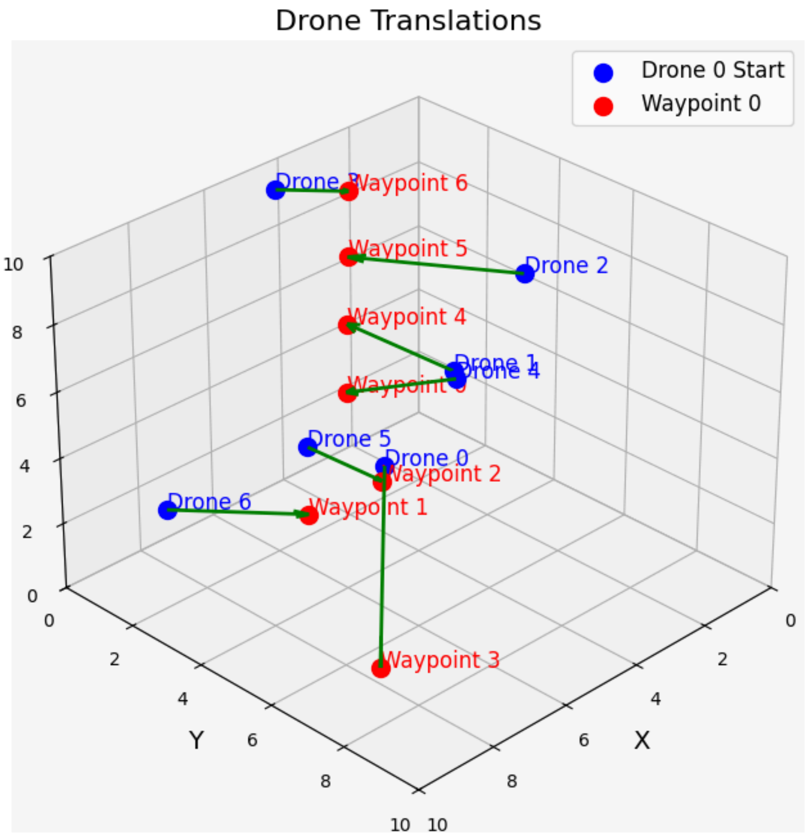
2. Test 2 (5 drones, optimal cost: 33.2249)



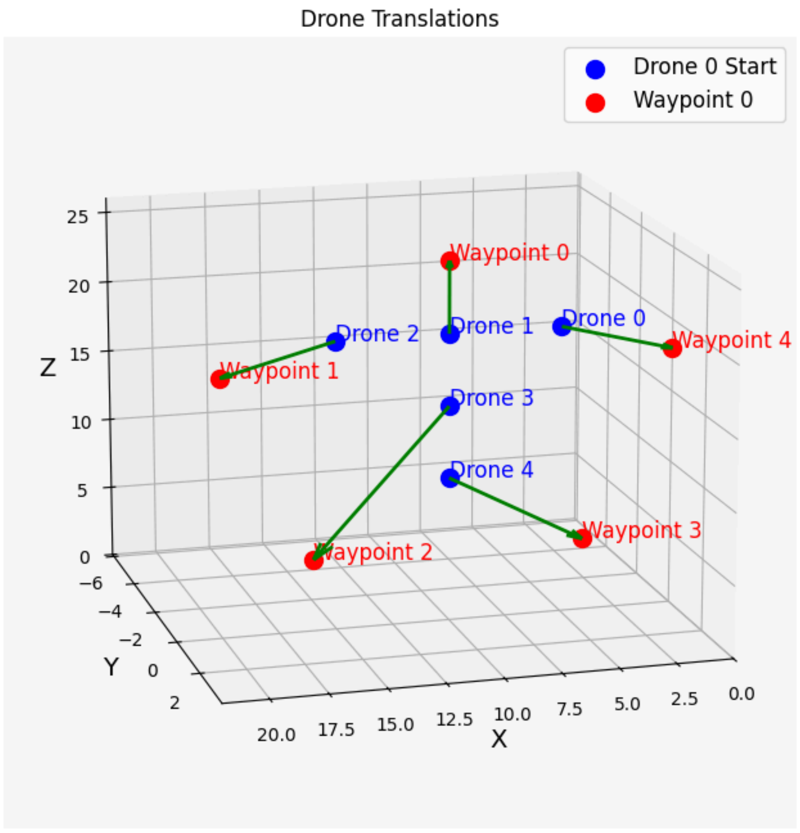
3. Test 3 (4 drones, optimal cost: 20.00)



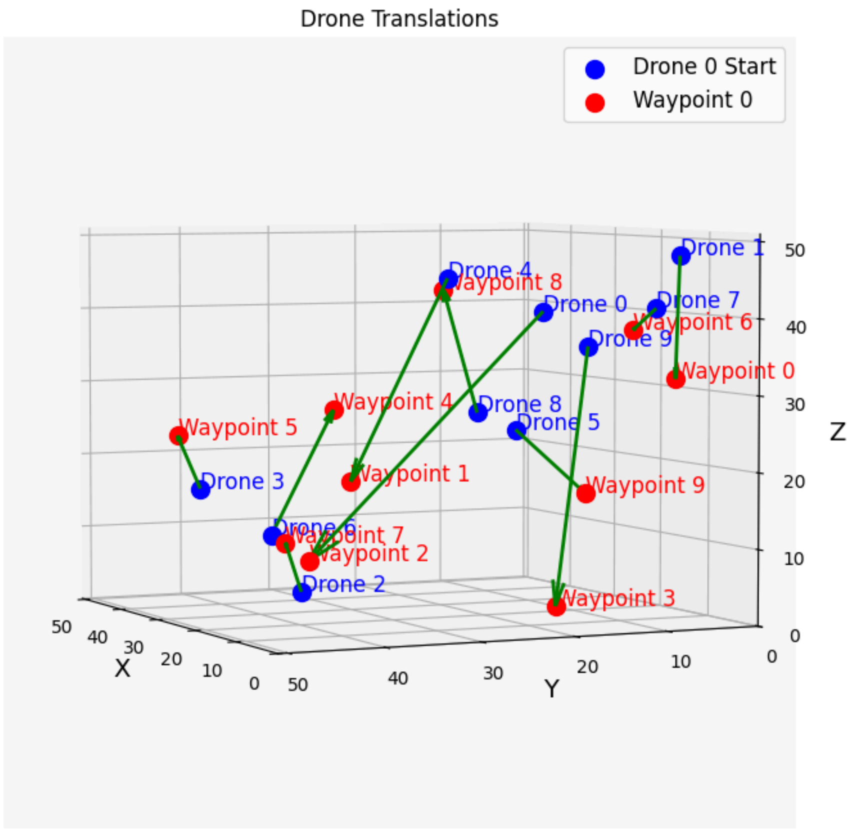
4. Test 4 (7 drones, optimal cost: 28.1417)



5. Test 5 (5 drones, optimal cost: 35.2425)

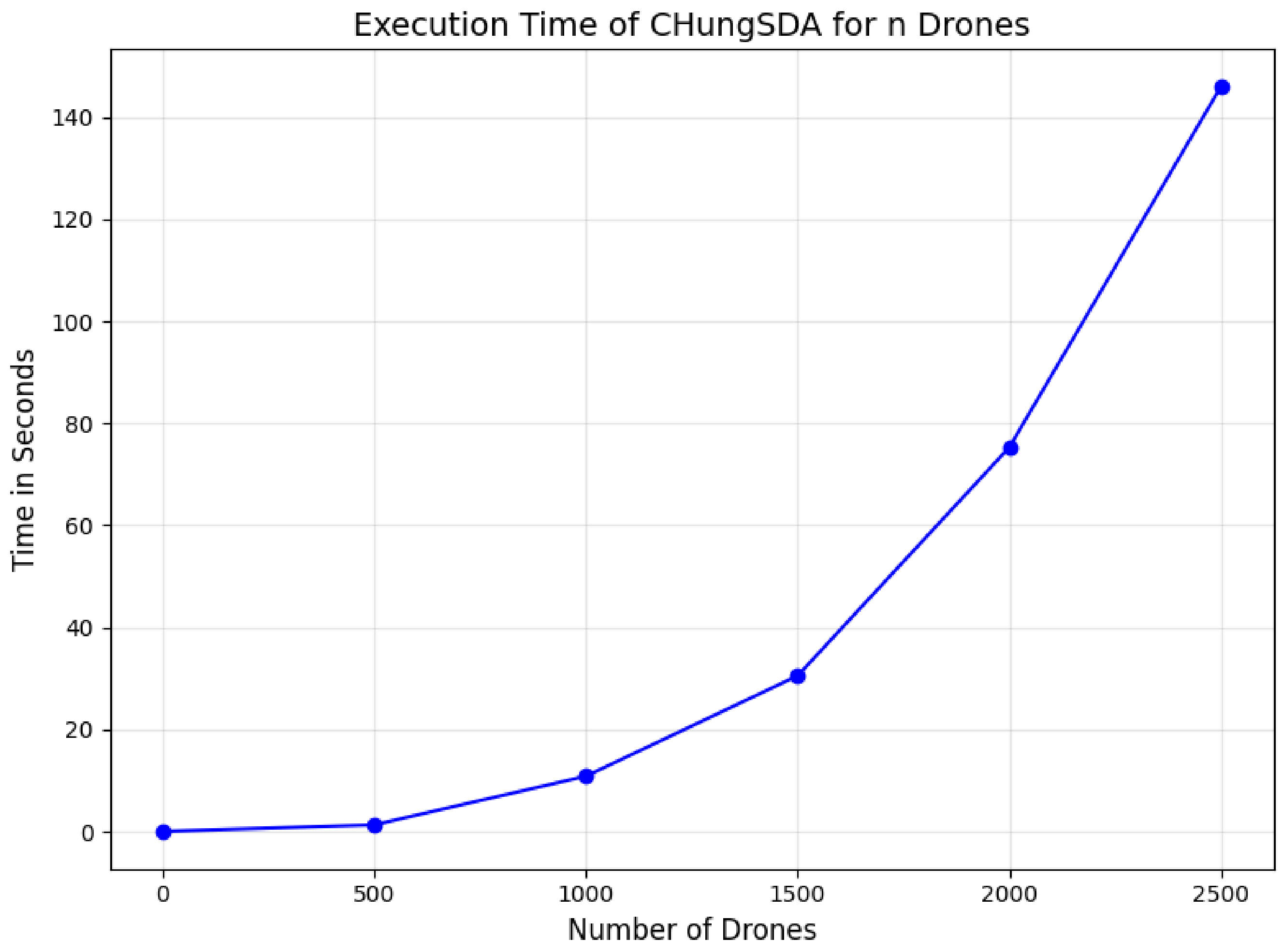


6. Test 6 (10 drones, optimal cost: 193.402)



Runtime

A graph was plotted with the runtime of the algorithm in seconds against the number of drones in the input. The results were as follows:



As stated earlier, this algorithm runs in $\mathcal{O}(n^3)$ time complexity, with at most n operations involving $\mathcal{O}(n^2)$ time calculations of the minimum number of lines to cover the zeros in each operation.

6. Future Work

1. Individual drone battery capacity constraint: While the current algorithm focuses on minimizing the total distance, it does not account for individual drone battery capacities—a critical practical consideration. Thus an additional constraint can be added into the algorithm to consider the maximum distance each individual drone can travel in a single flight.
2. An alternative optimization approach: while the total distance minimization is discussed in this paper, another potential approach could prioritize minimizing the time interval between consecutive transitions, which adds to the visual appeal of the show.
3. AI-based assignment: Wind conditions can alter the initial positions of the drones and hence their trajectories. This leads to the risk of close proximity and/or collisions. AI is essential to detect various such factors and also take into account any obstacles around.
4. The parallelization using OpenMP of the algorithm's working may be desirable in order that the problem can be solved in less amount of time for even greater number of drones, i.e., ≥ 5000.

7. Appendices

1. Pseudocode for CHungSDA

```
Function CHungSDA(n, D, W, deltaMin):
  Input: n (number of drones or waypoints),
         D (drone coordinates [x, y, z]), W (waypoint coordinates [x, y, z]),
         deltaMin (minimum separation distance between waypoints)
  Output: assignment (drone-to-waypoint pairs [u, v]), total_cost (number)

  # Step 1: Check distance constraints
  close_points = []
  For i = 1 to n:
    For j = i+1 to n:
      If euclidDist(W[i], W[j]) < deltaMin:
        Append (i, j) to close_points
  If close_points is not empty:
    Return "Constraint violated for", close_points

  # Step 2: Create and reduce cost matrix
  costMatrix = [[euclidDist(D[i], W[j]) for j in range(n)] for i in range(n)]
  For each row in costMatrix: Subtract row minimum
  For each column in costMatrix: Subtract column minimum

  # Step 3: Initialize variables
  let assignment = an empty list of pairs
  While true:
    rowCovered, colCovered = [False] * n, [False] * n
    VC_sets, edges = coverZeros(costMatrix)
    lines = len(VC_sets.first) + len(VC_sets.second)

    If lines == n:
      assignment = edges
      Break

    # Step 4: Adjust uncovered elements
    For i in VC_sets.first: rowCovered[i] = True
    For j in VC_sets.second: colCovered[j] = True
    min_uncovered = min(costMatrix[i][j]
                        for i, j
                        where rowCovered[i] == False
                        and colCovered[j] == False)
    For each i where rowCovered[i] == False:
            Subtract min_uncovered from costMatrix[i][*]
    For each j where colCovered[j] == True:
            Add min_uncovered to costMatrix[*][j]

  # Step 5: Calculate total cost
  total_cost = Sum(costMatrix[i][j] for (i, j) in assignment)
  Return assignment, total_cost
```

2. Pseudocode for coverZeros

```
Function coverZeros(matrix C):
  n = size of C

  # the following is an instance of an external module
  matching = bipartite_matching(n, n)

  # Add edges for all 0s in matrix
  For i = 0 to n-1:
    For j = 0 to n-1:
      If C[i][j] == 0:
        matching.add_edge(i, j)

  # Compute matching and vertex cover
  max_matching = matching.get_max_matching()
  VC_sets = matching.minimum_vertex_cover()
  edges = matching.get_edges()

  Return (VC_sets, edges)
```

3. C++ Code

Project GitHub Repository

4. References

Graph Matching (Wikipedia)
Code used for Bipartite Matching
Hungarian Algorithm Overview