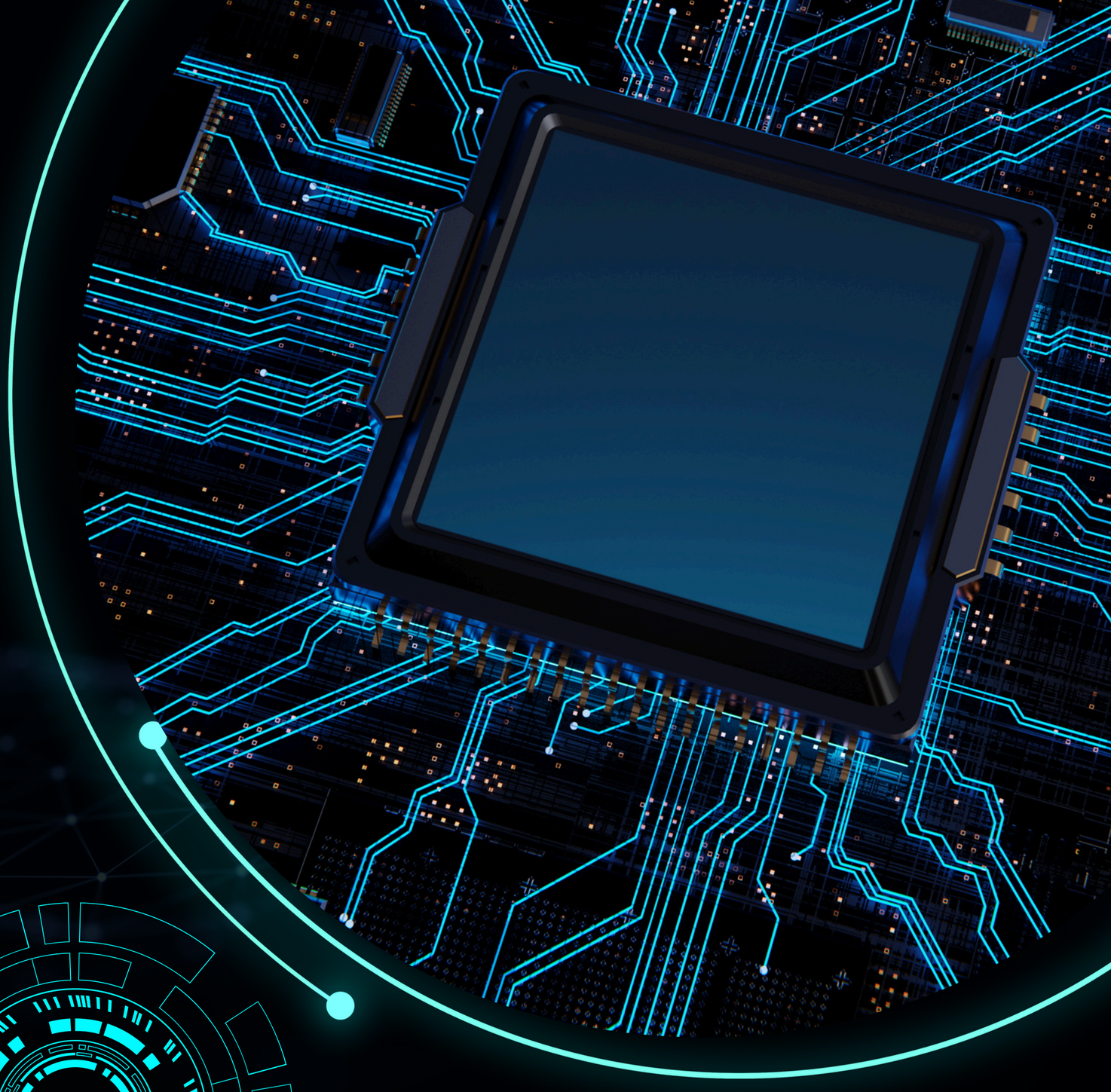
10 DAYS OVER KANPUR

TAKNEEK'25

PS Explained

Designing an efficient drone delivery system with optimized energy management, capable of handling diverse and dynamic orders, while adopting a careful strategy that ensures safe and timely deliveries, maximizes overall performance and score, and minimizes operational risks and penalties.



Our Solution → flowchart

DECRYPTION

Reverse the encryption algorithm to decrypt all the provided passwords

PRE-PROCESSING

Precomputes orders and graphs of charging points for faster processing

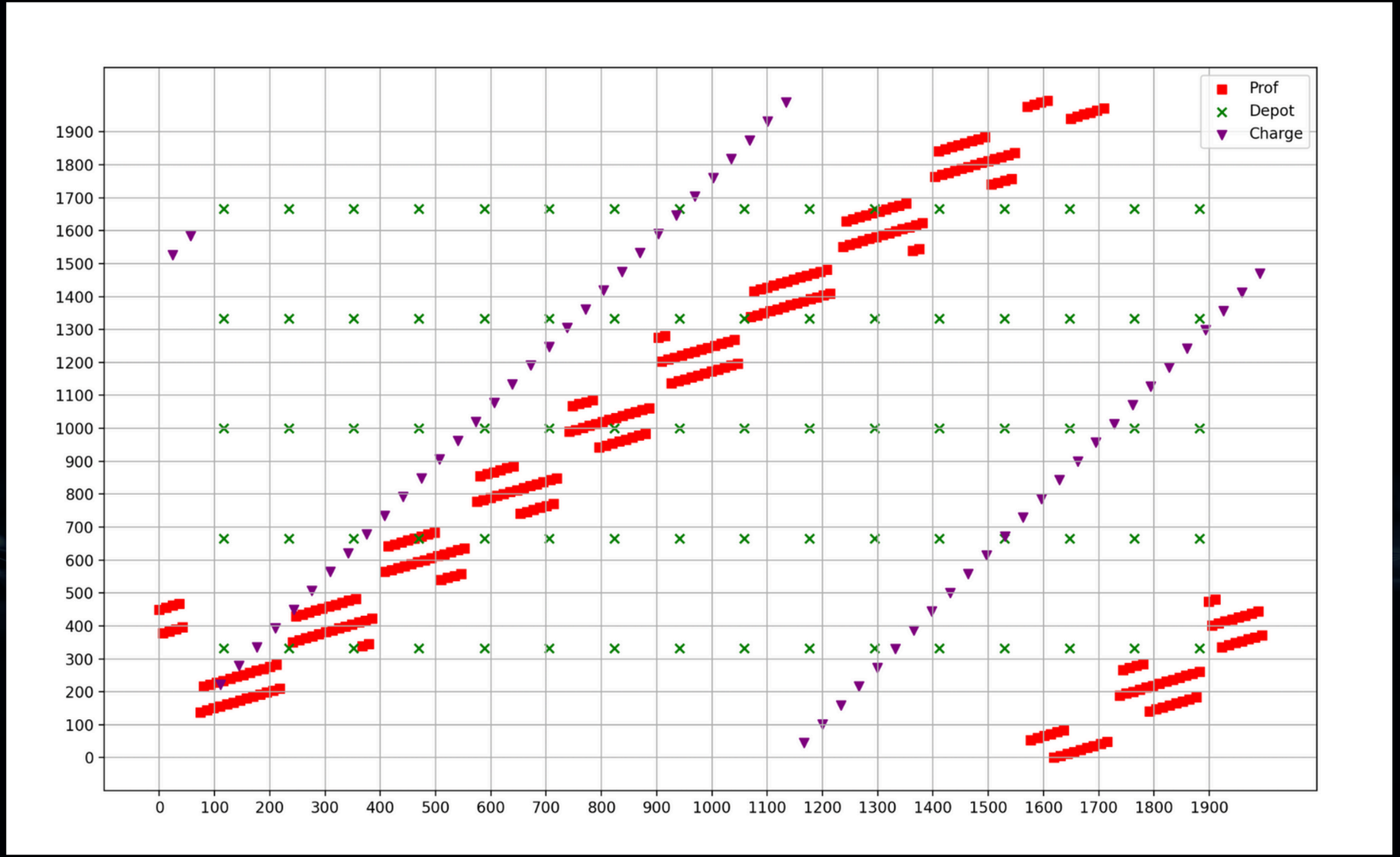
SIMULATION

Simulates the problem in turns and manages each drone independently.

VALIDATION

Validated the output file generated using a separate validator program to ensure no bugs in code.

Input Grid


Prof
Depot
Charge
0
100
200
300
400
500
600
700
800
900
1000
1100
1200
1300
1400
1500
1600
1700
1800
1900

Decryption Process

1. Generate primes:

$$p = a^4 + \text{secret}_1, \quad q = b^4 + \text{secret}_2,$$

where a, b are 512-bit integers and the secrets are small (16 bits).

2. Compute

$$n = p \cdot q \approx (ab)^4,$$

and estimate

$$k = \lfloor n^{1/4} \rfloor.$$

The true value of ab always satisfies $ab \geq k$.

3. Rewrite

$$n = (a^4 + \text{secret}_1)(b^4 + \text{secret}_2),$$

as a degree-8 polynomial in terms of a, and use `SageMath`'s `.roots()` method to find integer roots.

4. For each candidate a, compute p. If $p \mid n$, stop and compute $q = n/p$.

5. If no divisor is found, increment k and repeat for up to 1000 iterations.

6. Once p and q are recovered, calculate

$$\varphi(n) = (p-1)(q-1),$$

and compute the private key d from

$$ed \equiv 1 \pmod{\varphi(n)},$$

using `inverse_mod()` in `SageMath`.

7. Decrypt the ciphertext:

$$m \equiv c^d \pmod{n},$$

then convert m to binary and decode it into a string to obtain the final password.

High Level Approach

- **Our strategy:**

  Since the highest penalty is for losing drones, our solution revolves around charging stations. In the first part of our approach, we move all the drones to a secure position, i.e the nearest charging station.
- G**raph Construction**:

  Construct a graph between charging stations if the Manhattan distance is less than Bmax - 10.
  Use Floyd Warshall's algorithm to compute all pairs shortest paths.
  For every pair of depot and professor, we preprocess and find the best path to reach a professor from a depot using some sequence of charging stations.
- **Order Delivery**:

  Whenever a drone is assigned an order, it follows a safe path to reach the nearest charging station of the depot, the picks up the order by giving the correct password, follows the shortest path and when it reaches the charging station before professor, it stops and waits till the current turn becomes "appear_time of order - dist(professor, current position)".
  It charges till then, and then moves towards the professor, delivers the orders and moves to the nearest charging station. Then again, an order is assigned and the same process repeats.

Pre Processing in depth

Charging stations graph (chargingGraph()) -

Build initial adjacency by setting distCharge[i][j] to the Manhattan distance between charger coordinates when direct distance is < B_max - 20 (a heuristic threshold chosen in the solution). Run Floyd–Warshall to compute all-pairs shortest distances and populate pathCharge_next_node[i][j] to allow path reconstruction.

Build feasible depot→prof routes (DCPpreProcess()) -

For every depot d and professor c:-
Compute the optimal safe path with the least turns, ensuring no drone is lost in the path.
It follows a sequence of chargers to start from a depot d and reach professor c.
Here we store the starting and ending charger_id . Here we also make sure to only allow a virtual edge between professor/depot to chargers if the manhattan distance < B_max / 2 - 10 (about half the original heuristic threshold)

KEY IMPLEMENTATION PROCEDURE TURN BY TURN

Initial charging priority: the system attempts to get all drones to full battery at the start (or ensures drones are safely charged before assignments). This prevents early losses.

Assignment: When a drone is ready to take a job, it pops an order from the orderQueue and a path is assembled as:

current charger → (intermediate chargers along the charger path) → depot charger → depot → charger path to professor → professor → nearest charger

Execution: The above waypoint list is converted to concrete grid coordinates (charger/depot/prof coordinates) stored in dronePath[i]. Also the safety asserts check battery > 1 before issuing moves and >5 before pickup/dropoff.

Battery safety: If battery is low, the drone is routed to the nearest charger. Charging adds R battery units per turn up to Bmax .

KEY CONSTANTS AND TIME COMPLEXITY

Magic Numbers used:

- B_max - 20 and B_max/2-10 are distance thresholds used in charging station graph initialization and in feasibility filtering. These were the heuristics chosen to limit search and guarantee recharge paths.

Complexity:

- Expected Runtime - 15-30 seconds
- Charger graph: O(P^3) (Floyd–Warshall)
- Depot → Professor preprocessing: up to O(M * C * P^2)
- Order sorting: O(N log N)
- Simulation loop: O(T * D) (movement is O(1) per turn per drone).

Our solution is robust, delivering optimal efficiency with a simulation runtime of O(Turns × Drones) which is the best possible for this problem. The total runtime remains within 15-30 seconds, primarily due to essential preprocessing optimizations.

Possible Sources of Improvement

- The nearest charging station finding uses a naive brute force as the number of charging stations is quite less. For a higher number of charging stations, we could do a multi-source BFS to compute the same.

- Integrate inventory & order cancellation handling more robustly (race conditions at depots when stock runs out). Add better prediction for deadlines and dynamic re-routing when delays occur.


Thank You