Drone Fleet Optimization: Dynamic Delivery Planning in Highly Constrained Environments

FRANCKY RONSARD SAAH
TECHNOLOGY FACULTY
INFORMATION SYSTEM INGENEERİNG
KOCAELI, TURKEY
francky877832@gmail.com

KHADIM DIEYE
TECHNOLOGY FACULTY
INFORMATION SYSTEM INGENEERİNG
KOCAELI, TURKEY
Khadimd978@gmail.com

TURAN ASGARLI
TECHNOLOGY FACULTY
INFORMATION SYSTEM INGENEERİNG
KOCAELI, TURKEY
turanaskerov2004@gmail.com

Abstract- *In recent years,* ***drone delivery systems*** *have gained significant attention as a solution to last-mile logistics challenges. However,* ***real-world applications*** *are constrained by multiple dynamic factors such as limited* ***energy capacity****, airspace restrictions (no-fly zones),* ***delivery priority****, and drone availability. This paper presents a hybrid approach for drone delivery route planning under such constraints by combining the* ***A* search algorithm, Constraint Satisfaction Problems (CSP), and Genetic Algorithms (GA)****. A graph-based environment is modeled with dynamically updated nodes representing delivery points and edges reflecting cost penalties for energy consumption and delivery priority. A* is utilized to compute optimal paths between delivery nodes while avoiding restricted zones. A* ***CSP*** *module ensures that generated plans adhere to key constraints, and a GA is employed to optimize delivery sequences across multiple drones. Experimental results show that this hybrid method effectively maximizes delivery success rate while minimizing energy consumption and constraint violations, offering a promising solution for autonomous delivery route planning in constrained urban environments.*

Keywords - *Drone Delivery, real-world applications, . energy capacity, delivery priority, A* search algorithm, Constraint Satisfaction Problems, and Genetic Algorithms, CSP.*

1. Introduction

The increasing demand for fast, cost-effective, and autonomous delivery systems has fueled the development of drone-based logistics networks. Drones offer a promising alternative to traditional ground transport by navigating directly through airspace, significantly reducing travel time. Despite this potential, deploying drones in real-world delivery scenarios introduces a set of complex and dynamic constraints. These include limited battery life, weight restrictions, time-sensitive deliveries, and environmental constraints such as no-fly zones or weather conditions.

Efficiently assigning delivery tasks to a limited fleet of drones and planning their routes requires intelligent algorithms capable of handling dynamic variables and making trade-offs between optimality and feasibility. Traditional pathfinding methods may struggle when the solution space becomes heavily constrained or dynamically changes over time.

This paper proposes a hybrid system for constrained drone delivery route planning using three core algorithmic strategies: A* for pathfinding, CSP for constraint enforcement, and GA for global optimization. The system first uses CSP to assign deliveries in a valid way considering battery, zone, and assignment constraints. Then, A* calculates the optimal path from the drone's current location to its assigned delivery location, accounting for energy cost and no-fly zones. Finally, a genetic algorithm improves overall system performance by evolving delivery assignments and sequences based on a multi-objective fitness function. The goal is to maximize the number of successful deliveries while minimizing energy usage and constraint violations.

This hybrid approach is implemented and tested in a simulated environment based on a sparse graph, representing delivery points and their connectivity. The experimental setup evaluates the effectiveness of the proposed system in scenarios involving multiple drones and dynamically changing constraints.

2. Problem Definition

This project addresses the problem of optimal route planning for a fleet of drones operating under multiple dynamic constraints. The goal is to assign and execute deliveries such that total efficiency is maximized while respecting operational limitations.

a. Objective

A logistics company aims to deliver packages of varying weight and priority using a fleet of autonomous drones. The system must compute valid and efficient delivery routes that:

Maximize the number of completed deliveries,

Minimize total energy consumption,

Avoid restricted airspace (no-fly zones),

Respect individual drone capacity constraints,

Honor time windows and delivery priorities.

b. Constraints and Data Structures

The input data includes drones, deliveries, and no-fly zones. These are represented as structured objects parsed from a JSON scenario file.

Drones

Each drone is defined by:

id: Unique identifier.

max_weight: Maximum payload capacity in kilograms.

battery: Battery capacity in mAh.

speed: Speed in meters per second.

start_pos: Starting location as an (x, y) coordinate.

Deliveries

Each delivery task includes:

id: Unique identifier.

pos: Delivery location as an (x, y) coordinate.

weight: Package weight in kilograms.

priority: Priority level (1: low to 5: high).

time_window: Acceptable time interval for delivery.

No-Fly Zones

Each restricted zone is described by:

id: Unique identifier.

coordinates: Polygon vertices defining the zone.

active_time: Time interval during which the zone is restricted.

c. Challenges

The problem involves both *assignment* (which drone should perform which delivery) and *routing* (what path should be followed). Major challenges include:

Dynamic Constraints: Time-sensitive restrictions such as no-fly zones that activate at certain hours.

Resource Limitations: Drones have limited battery capacity and can carry only one package at a time.

Complex Cost Metrics: Delivery cost depends on distance, energy use, and delivery urgency.

3. Algorithm Design

To solve the route planning problem under multiple constraints, a hybrid approach is employed combining graph search (A*), constraint satisfaction (CSP), and genetic algorithms (GA). The overall architecture includes the following components:

a. Graph Construction

The delivery area is modeled as a graph where:

Nodes represent delivery locations.

Edges represent potential travel paths between these locations.

Multiple types of graphs can be generated depending on the scenario:

Complete Graph: Each node is connected to every other node.

Sparse Graph: Each node is connected to a limited number of neighbors (e.g., 3).

Oriented Graphs: Connections are directional, simulating one-way air corridors.

Cost Function for Edges

The cost of an edge between two delivery points iii and jjj is computed using:

Maliyet fonksiyonu (cost) = distance × weight + (priority × 100)

This function increases with travel distance, payload weight, and delivery urgency, penalizing high-priority deliveries that are delayed or inefficiently routed.

b. Route Search with A* Algorithm

A* is used to find the shortest valid path between two points in the graph, taking into account dynamic and static constraints.

Heuristic Function

$$f(n) = g(n) + h(n)$$

Where:

g(n) is the actual cost from the source to node nnn,

h(n) = distance + nofly_zone_penalty is the heuristic estimate to the goal, including: *Euclidean distance*, *Penalty if a path intersects an active no-fly zone.*

The algorithm discards paths that exceed the drone's weight capacity, require more battery than the drone can provide. And Penalize those that enter active no-fly zones during restricted times,

c. Constraint Satisfaction Problem (CSP)

Before optimization, a CSP model ensures that any assignment of deliveries to drones is feasible. The CSP phase checks:

- One drone can carry only one delivery at a time.
- The total weight of assigned deliveries does not exceed the drone’s capacity.
- R restricted areas are penalized.
- Time windows for deliveries are respected.

CSP filtering prevents invalid initial populations from being passed to the GA phase.

c. Genetic Algorithm for Optimization

Once a valid population is generated, a genetic algorithm is used to evolve delivery schedules toward optimality.

Initial Population - generated using:

Randomly valid assignments based on the CSP check.

Each individual in the population is a mapping from drones to delivery sequences.

Genetic Operators

Crossover: Combines delivery assignments of two parent individuals to create new offspring.

Mutation: Randomly reassigns a delivery to a different drone or modifies delivery order.

Selection: Tournament selection is used to choose parents for the next generation.

Fitness = (completed delivery × 50) – (total energy × 0.1) (violations× 1000)

This function rewards individuals that complete more deliveries efficiently and penalizes those that violate constraints such as weight limits, time windows, or flight restrictions.

4. Project Structure

```
drone/
├── algorithms/
│   ├── __pycache__/
│   │   ├── a_star.python-3.11.pyc
│   │   └── graph_builder.python-3.11.pyc
│   ├── a_star.py
│   └── graph_builder.py
├── assets/
│   ├── a_star_algorithm_flowchart.png
│   ├── a_star_algorithm_flowchart.svg
│   ├── a_star_vs_ga_diagram.png
│   ├── a_star_vs_ga_diagram.svg
│   ├── evaluate_individual_flowchart.png
│   ├── evaluate_individual_flowchart.svg
│   ├── generate_random_flowchart.png
│   ├── generate_random_flowchart.svg
│   ├── program_workflow_flowchart.png
│   └── program_workflow_flowchart.svg
│
```

```
├── data/
│   └── sample_data.txt
├── ga/
│   ├── __pycache__/
│   ├── fitness.py
│   ├── ga.py
│   └── population.py
├── graphics/
│   ├── __pycache__/
│   └── graph.py
├── models/
├── tests/
│   ├── test_a_star.py
│   └── test_route_planning.py
├── utils/
│   ├── __pycache__/
│   ├── constraints.py
│   ├── helpers.py
│   ├── scenario_1_generator.py
│   └── scenario_2_generator.py
│   ├── simulate_delivery.py
├── main.py
├── scenario1.json
└── scenario2.json
```

5. Project Main Alogirthms Explanation

a. ***a_start algorithm (algoritms/a_start.py)*** *– the a_start function algorithm to find th less cost path (optimal path)for the current delivery.*

The a_star function is an implementation of the A* search algorithm used to find the optimal path from a start position to a goal position, while considering drone-specific constraints and no-fly zones. The A* algorithm is a well-known pathfinding technique that uses a combination of actual movement cost and a heuristic to estimate the best possible path to the goal.

The algorithm is implemented as follows:

Function Input and Setup

The function takes several inputs:

graph: A dictionary representing the connections between nodes (locations) with their associated costs.

start: The starting position of the drone.

goal: The target destination of the drone (e.g., a delivery location).

nofly_zones: A list of no-fly zones that the drone should avoid.

drone: An object representing the drone, which contains attributes such as its maximum weight capacity and battery life.

deliveries: A list of delivery locations the drone must visit.

The algorithm starts by converting the positions of the start and goal into tuples and sets up an initial configuration for the search process.

Heuristic Function

The heuristic function calculates an estimate of the remaining cost to reach the goal from the current node. It is a key component of the A* algorithm:

Euclidean Distance: The straight-line distance from the current node to the goal is calculated using euclidean_distance(n, target).

Penalty for No-Fly Zones: If the current node is within a no-fly zone, the penalty is added using the apply_penality(n, target, nofly_zones) function.

The heuristic returns the sum of the distance and the penalty, guiding the search toward the goal while avoiding restricted areas.

Path Reconstruction

Once the goal is reached, the redraw_path function is used to reconstruct the optimal path from the goal back to the start:

It iteratively traces back from the goal to the start by following the parent nodes stored in the came_from dictionary.

If no path is found (i.e., the goal cannot be reached), it returns an empty list.

Main A Search Process*

The main search loop is initiated by pushing the starting position into the open_set priority queue. The queue is implemented as a heap, where nodes are stored with their associated costs (f_score), actual movement cost (g_score), and their position.

The search proceeds as follows:

Node Exploration: The node with the lowest f_score (estimated total cost) is extracted from the open set. If the current node matches the goal, the optimal path is reconstructed and returned.

Neighbor Exploration: The neighbors of the current node are evaluated:

Each neighbor's weight must not exceed the drone's maximum weight capacity.

The battery required for the round-trip to the neighbor is calculated based on Euclidean distance, and it must not exceed the drone's available battery.

A penalty is added for traversing no-fly zones.

If the neighbor has not been visited yet or if a better path to it is found, it is added to the open set with an updated cost.

Pathfinding Termination

If the open set becomes empty, meaning no valid path exists, the function returns None indicating that no feasible path was found. If a path is found, it is returned as a list of positions that represent the optimal route from the start to the goal.

b. Graph_builder.py

This file contains several function to create different type of graphes.

build_graph *- Full Graph of All Points*

The build_graph function generates a **full graph** that includes all the delivery points and the starting point:

It creates a graph where each point (either start or a delivery) is connected to every other point.

Output: A complete graph where each node is connected to every other node, and the cost to travel between them is stored.

This type of graph can be useful for scenarios where it's important to consider all possible routes, but it can be computationally expensive as the number of delivery points grows.

generate_complete_graph *- Complete Graph with Delivery Points*

The generate_complete_graph function creates a **complete graph** of delivery points:

The graph connects each delivery point to every other delivery point, calculating the cost using the compute_base_cost function for each pair.

Output: A fully connected graph of delivery points with associated travel costs.

generate_oriented_sparse_graph *- Sparse Directed Graph with k Nearest Neighbors*

The **generate_oriented_sparse_graph** function generates a **sparse directed graph** where each node (delivery point) only connects to its k nearest neighbors:

For each delivery point, the algorithm computes the Euclidean distance to all other points and sorts them by distance.

It then keeps only the k nearest neighbors for each node, creating directed edges from each node to its k nearest neighbors.

Output: A sparse, directed graph where each node has edges pointing to its k nearest neighbors, with associated movement costs.

This approach reduces the complexity of the graph by limiting connections to the nearest points. It's suitable for large datasets where keeping all possible connections is inefficient, and the focus is on the most relevant neighbors.

generate_sparse_graph *- Sparse Undirected Graph with k Nearest Neighbors*

The generate_sparse_graph function builds a **sparse undirected graph**:

Similar to the **generate_oriented_sparse_graph**, it computes the Euclidean distance between all points and keeps the k closest neighbors.

Unlike the oriented graph, this version adds bidirectional (undirected) edges between each point and its nearest neighbors. Both directions of travel between points are considered.

Output: A sparse, undirected graph where each node is connected to its k nearest neighbors.

This type of graph is useful when the relationships between points are bidirectional (i.e., travel is possible in both directions with the same cost). It also reduces the graph’s size by only considering the nearest neighbors.

Application in Drone Delivery

These graphs are used to model the movement of drones between delivery points. By considering different types of graphs, the system can optimize route planning by balancing between computation efficiency and the accuracy of the model. Sparse graphs are especially helpful in reducing the computational load in large-scale environments, while full graphs ensure that all possible routes are considered when needed.

c. Evaluate_individual(ga/fitness.py)

Evaluation Function for Genetic Algorithm: evaluate_individual

The evaluate_individual function is a key part of the genetic algorithm used to optimize drone delivery routes. This function calculates the fitness of a given individual (i.e., a specific delivery sequence assigned to each drone) based on various factors, such as energy consumption, constraint violations, and successful deliveries.

Purpose

The purpose of the evaluate_individual function is to assess the quality of a candidate solution in the genetic algorithm by simulating the delivery process for each drone and calculating relevant performance metrics. It takes into account the energy used by the drone, the number of violations (e.g., due to time windows or constraints), and the number of successful deliveries.

Inputs

individual: A dictionary where each key is a drone identifier (e.g., "D1", "D2") and each value is a list of delivery IDs representing the sequence of deliveries assigned to that drone.

graph: The graph representing the environment, where nodes are delivery points, and edges are the costs (e.g., distances or time) between those points.

no_fly_zones: A list of restricted zones where the drone is not allowed to fly.

drones: A list of drone objects, each representing a drone with attributes such as its ID, battery, speed, and other specifications.

deliveries: A list of delivery objects, each containing information such as the delivery point's position, weight, and priority.

Process

Initialize Variables:

total_energy: Keeps track of the total energy consumed by all drones.

total_violations: Tracks the number of constraint violations (e.g., if the drone exceeds weight limits, violates time windows, etc.).

successful_deliveries: Counts the number of successful deliveries made.

Iterate Over Each Drone and Delivery Sequence:

For each drone in the individual’s delivery sequence:

Reset Drone: The drone's state (e.g., battery, weight) is reset to its initial values using drone.reset().

Iterate Over Deliveries: For each delivery in the drone's sequence:

Pathfinding: The a_star algorithm is used to calculate the optimal path for the drone between its current position and the goal (delivery location).

Cost Calculation: The cost of traveling along the path is calculated based on the graph.

Constraint Checking: The function check_all_csp_for_violation is used to check if the drone violates any constraints (e.g., battery, weight, or time window).

Simulate Delivery: The drone moves to the delivery location, updates its battery, and the delivery is marked as complete.

Recharging: After completing a delivery, the drone starts recharging, and the estimated arrival time is computed based on the drone's speed and the distance traveled.

Fitness Calculation:

The fitness of the individual is computed as:

Fitness = (number of deliveries × 50) – (total energy × 0.1) – (violated constraints × 1000)

Successful deliveries are rewarded with 50 points each.

Energy consumption is penalized by 0.1 per unit of energy used.

Violations are heavily penalized with a high cost (1000 per violation) to discourage solutions that violate constraints.

Return Fitness: The fitness score is returned, representing the quality of the delivery sequence for the drone.

Outputs

fitness: The fitness score of the individual (i.e., the delivery sequence). A higher fitness score indicates a better solution, with more successful deliveries, less energy consumption, and fewer violations.

Application in Drone Delivery Optimization

This evaluation function is integral to the genetic algorithm, where each "individual" (delivery sequence) is scored based on how well it performs in terms of energy efficiency, constraint adherence, and the number of successful deliveries. The algorithm uses this fitness score to guide the evolution of better solutions by selecting, crossing over, and mutating individuals to find optimal delivery routes for the drones.

By incorporating factors like energy consumption, constraint violations, and delivery success, the system ensures that the resulting routes are not only feasible but also efficient and compliant with operational constraints (such as battery life, weight limits, and time windows).

d. Ga/ga.py

Crossover Function: crossover

The crossover function combines two parent solutions (delivery sequences) to create offspring. It randomly selects a crossover point, swaps the deliveries after this point between the parents, and then reconstructs the offspring's delivery sequence. This is a form of recombination in genetic algorithms that helps explore new solutions.

Inputs: Two parent dictionaries representing delivery sequences.

Outputs: Two offspring dictionaries, each representing a new delivery sequence.

Mutation Function: mutate

The mutation function introduces small changes to an individual solution. It randomly selects a drone and a delivery, removes a delivery, and replaces it with another available delivery. This helps maintain diversity within the population and prevents the algorithm from converging too quickly to suboptimal solutions.

Inputs: The individual (a dictionary of drone assignments) and the list of deliveries.

Outputs: The mutated individual (updated drone assignments).

Tournament Selection Function: tournament_selection

This function selects the best individual from a randomly chosen subset of the population. The selection is based on the fitness of the individuals, which is calculated using the evaluate_individual function. This ensures that the best-performing individuals are more likely to be selected for reproduction.

Inputs: The population, graph, no-fly zones, drones, deliveries, and tournament size.

Outputs: The best individual from the tournament.

Generate Next Generation: generate_next_generation

The function generates the next generation of individuals. It uses tournament selection to pick parents, performs crossover to create offspring, and applies mutation with a certain probability. The new generation is created by combining the best individuals and introducing new genetic material through crossover and mutation.

Inputs: The population, graph, no-fly zones, drones, deliveries, and mutation probability.

Outputs: The new population (next generation of individuals).

Ga/population.py

Generate Random Individual: generate_random_individual

This function creates a random individual (solution) by assigning each delivery to a drone. The deliveries are shuffled, and each drone is assigned one delivery (ensuring that no drone carries more than one delivery). The function checks each drone's assignments to ensure no drone is assigned multiple deliveries.

Inputs: List of drones and deliveries.

Outputs: A random individual, represented as a dictionary where keys are drone IDs, and values are lists of delivery IDs.

Generate Random Full Individual: generate_random_full_individual

This function generates a random individual with the added constraint that a drone cannot deliver to its starting position. If the shuffled deliveries contain a delivery assigned to the drone's starting position, the function retries until all drones are assigned valid deliveries. It ensures that no drone delivers to its own start position and that all drones are assigned at least one delivery.

Inputs: List of drones and deliveries.

Outputs: A random full individual, ensuring that each drone is assigned a valid delivery.

Generate Initial Population: generate_initial_population

This function creates an initial population of individuals. It uses the generate_random_individual function to generate a list of random individuals. The number of individuals in the population is defined by the size parameter.

Inputs: List of drones, deliveries, and the population size (default is 5).

Outputs: A list of random individuals, each representing a possible solution.

Generate Initial Full Population: generate_initial_full_population

This function generates an initial population of individuals where each individual adheres to the constraint that no drone delivers to its starting position. It uses the generate_random_full_individual function to create the individuals. The population size is defined by the size parameter.

Inputs: List of drones, deliveries, and the population size (default is 5).

Outputs: A list of random full individuals, each respecting the drone constraints.

Why Integrate generate_initial_full_population**?**

In the initial design of the algorithm, drones are initialized with their starting positions, and each drone is assigned a delivery from the list of available deliveries. However, this approach led to a critical issue: **some drones were assigned deliveries from the same point where they started**. For example, if drone 1 had a starting position at delivery.pos, it would end up trying to deliver to the point where it is already located. This is a **logical flaw**, as a drone cannot deliver to its own starting position.

To solve this problem and ensure **valid assignments**, we integrated the **generate_initial_full_population function**. This function performs additional checks to ensure that no drone is assigned a delivery where it starts. If such an assignment occurs, it retries until a valid assignment is made. The result is that every drone in the initial population is guaranteed to have a **valid delivery assignment**, adhering to the constraint that a drone cannot deliver to its own starting position.

How generate_initial_full_population **Works**

Prevention of Invalid Assignments: The function checks whether the delivery assigned to a drone is from its starting position. If it finds that a drone is about to deliver to its own starting position, it reshuffles the deliveries and retries the assignment.

Ensures All Drones Are Assigned Deliveries: The function ensures that each drone is assigned a delivery and no drone is left idle.

Improved Population Validity: With this function, the generated population consists of fully valid individuals, reducing the risk of generating invalid solutions that might need to be discarded during the evolution process.

By using this approach, we avoid the computational overhead of checking and discarding invalid individuals after they've been generated, leading to a **more efficient genetic algorithm**.

e. Graphics/graphc.py

plot_combined_graph_and_path()

This function plots a **combined graph** that includes:

No-fly zones: These are plotted as red polygons (with transparency) on the map.

Delivery points: Plotted as blue dots with IDs next to them.

Graph edges: Represent the connections between delivery points.

Path: A sequence of coordinates representing the drone's planned route, plotted in blue with arrows to indicate direction.

plot_graph()

This function is similar to the first one but without the path plotting. It focuses on:

No-fly zones: As red polygons.

Delivery points: Plotted as blue dots with IDs.

Graph edges: The connections between delivery points in gray.

plot_combined_oriented_graph_and_path()

This function is similar to the plot_combined_graph_and_path but with the added feature of **arrow indicators** on the edges of the graph to show the **direction of movement**. The edges are drawn with small arrows indicating the travel direction between points. This is helpful for visualizing the route in an oriented graph.

plot_oriented_graph()

This function plots the graph with:

No-fly zones.

Delivery points.

Directed edges: Each edge between nodes (delivery points) has a small arrow to indicate direction.

plot_path() : initially to test a_start

This function visualizes the drone's **path**:

Path plotted with markers: The path is drawn with blue circles at each point.

Arrows for direction: Arrows are added along the path to show movement direction.

Labels: Delivery IDs and coordinates are labeled at each point in the path.

Why the Integration of Arrows and Directions?

The direction arrows in the last three functions allow better tracking of the drone's movement along the route. This can be particularly useful when:

Visualizing delivery routes where the drone needs to follow a specific path.

Identifying the order of delivery execution.

Understanding connectivity in directed graphs (important for algorithms like A* or pathfinding).

These visualization functions give you a clearer picture of the drone’s planned route, **no-fly zones**, and the **graph structure** of delivery points, which aids in analysis and debugging.

f. Constrainst/constrains.py

This code is focused on **Constraint Satisfaction Problem (CSP)** checks for drone deliveries. It checks whether a drone can successfully complete a given delivery based on various conditions, such as its availability, battery level, maximum weight capacity, and whether it can meet the delivery's time window. Let's break it down:

Functions Overview:

check_single_delivery_per_trip(drone)

Purpose: Checks if the drone is available for a new delivery. It uses the drone.is_available() method, which likely checks if the drone is currently assigned to another task.

check_drone_recharging(drone)

Purpose: Checks if the drone is currently recharging. It uses the drone.is_recharging attribute.

check_drone_can_support_cost(drone, needed_cost)

Purpose: Checks if the drone has enough battery to complete the delivery. The method compares the drone's battery with the needed_cost (likely the energy required for the delivery).

check_drone_can_support_weigth(drone, delivery)

Purpose: Checks if the drone can carry the delivery based on its maximum weight capacity. The method compares the drone's max_weight with the delivery's weight.

chech_drone_is_within_time_window(estimated_arrival_time, delivery)

Purpose: Checks if the drone's estimated arrival time at the delivery point falls within the delivery's time window. The is_within_time_window() method is used here, which likely checks if the time falls between the start and end of the delivery's time window.

check_all_csp(start, goal, drone, delivery, needed_cost)

Purpose: This is the main CSP check function. It performs a series of checks before assigning a drone to a delivery. If all conditions are met (i.e., no CSP violations), it returns the drone; otherwise, it prints the violation type and returns None. It checks:

If the drone is available for the trip.

If the drone is not recharging.

If the drone has enough battery for the delivery.

If the drone can carry the delivery's weight.

If the drone can arrive within the delivery's time window.

check_all_csp_for_violation(start, goal, drone, delivery, needed_cost)

Purpose: Similar to check_all_csp, but instead of returning the drone, it counts how many CSP violations occur for a given drone and delivery. The function increments the violation count for each failed condition (e.g., drone is unavailable, insufficient battery, etc.) and returns the total number of violations.

The **violation tracking** function (check_all_csp_for_violation) helps in diagnosing why a drone cannot complete a task by counting the number of issues.

Example Workflow:

A drone is evaluated for a delivery by the check_all_csp function before being assigned.

If the drone meets all conditions (availability, battery, weight capacity, time window), it is selected for the delivery.

If not, it is **re-added to the queue** and a violation message is printed, detailing the issue (e.g., insufficient battery, exceeding weight, outside time window).

By integrating this into the drone delivery system, you can ensure that only eligible drones are selected for tasks, thereby optimizing your system's performance and reliability.

g. Utilities fnctions

initialize_drones_on_graph(delivery_points, drones)

Randomly assigns each drone a starting position from a list of delivery points.

compute_base_cost(pos1, pos2, weight, priority)

Calculates the base cost for a delivery route based on distance, weight, and priority.

euclidean_distance(p1, p2)

Computes the Euclidean distance between two points p1 and p2.

heuristic(n, target, nofly_zones)

Estimates the cost to reach a target node from a current node, including penalties for passing through no-fly zones.

apply_fixed_penality(start_node, end_node, nofly_zones)

Returns a fixed penalty if the movement from start_node to end_node intersects a no-fly zone.

apply_penality(start_node, end_node, nofly_zones, cost_per_meter=fixed_penality)

Calculates a penalty based on the length of the path through no-fly zones, where the penalty depends on the zone's intersection with the path.

compute_cost(start_node, end_node, weight, priority, nofly_zones)

Computes the total cost for a route from start_node to end_node, considering distance, weight, priority, and penalties for no-fly zones.

is_within_time_window(current_time_str, time_window)

Checks if the current time is within the specified time window.

get_battery_needed(drone, delivery)

Calculates the battery required for a drone to complete a round-trip to a delivery point.

get_neighbors(current_node, graph)

Returns a list of neighboring nodes and their associated costs in the graph.

is_valid_move(start_node, end_node, nofly_zones)

Checks if the move between start_node and end_node does not pass through a no-fly zone.

is_within_no_fly_zone(nx, ny, no_fly_zones)

Determines if a point (nx, ny) lies within any no-fly zone.

has_capacity_for_delivery(drone, delivery)

Checks if a drone has the capacity to carry the delivery's weight.

used_energy(distance, weight)

Calculates the energy consumed based on the distance traveled and the weight of the delivery.

has_enough_battery_for_move(drone, nx, ny)

Checks if the drone has enough battery to make the move to a new point (nx, ny).

estimate_arrival_time(start_time_str, distance, speed)

Estimates the arrival time of a drone based on its start time, distance, and speed.

These functions provide various utilities for managing drone deliveries, from calculating costs and penalties to checking drone availability and battery usage.

h. Scenario Generator

This Python script generates a scenario for drone deliveries, including drones, deliveries, and no-fly zones. Here's a breakdown of the script's functionality:

Scenario Parameters:

scenario_no: Scenario identifier.

num_drone: Number of drones.

num_delivery: Number of deliveries.

delivery_timedelta_hours: Duration for which deliveries are active.

de**livery_hours_start & delivery_hours_end**: Time window for deliveries (from 08:00 to 18:00).

num_nfz: Number of no-fly zones.

nfz_hours_start & nfz_hours_end: Time window during which no-fly zones are active.

no_fly_zones_vertices: Number of vertices for defining no-fly zones (generating square/rectangular zones).

Drone Properties:

Drones have random attributes for weight, battery capacity, speed, starting position, and start time.

drone_min_weight and drone_max_weight define the weight range drones can carry.

drone_min_battery and drone_max_battery define the battery capacity range.

drone_min_speed and drone_max_speed define the speed range for the drones.

Delivery Properties:

Each delivery has a random position, weight, priority, and time window.

The weight is randomly generated between delivery_min_weight and delivery_max_weight.

No-Fly Zones:

Randomly generates no-fly zones as polygons defined by vertices and an active time window.

Saving the Scenario:

The scenario (drones, deliveries, and no-fly zones) is saved as a TXT file.

Key Functions:

random_coord(): Generates a random coordinate (x, y) within a specified range (x_max, y_max).

random_time_window(): Generates a random time window for a delivery, with a start time between 08:00 and 18:00, and a duration of delivery_timedelta_hours.

Outputs:

A TXT file named **scenario1.txt** (or another name depending on the scenario_no) is created with the generated data for drones, deliveries, and no-fly zones.

This script helps simulate a scenario with multiple drones, deliveries, and no-fly zones, which can be used for testing drone delivery algorithms.

6. How All These Work Together?

The project runs in main.py

Explanation of the Algorithm

This algorithm is designed to simulate a drone delivery system using Genetic Algorithms (GA) for route optimization. Here's a breakdown of the steps involved:

a. **Data Initialization:**

The data is loaded from a .txt file (scenario1.txt) which includes information about drones, deliveries, and no-fly zones.

The drones, deliveries, and no-fly zones are instantiated as respective objects (Drone, Delivery, NoFlyZone).

b. **Drone Initialization:**

Drones are initialized on a graph using their starting positions and delivery points.

c. **Graph Generation:**

Different types of graphs are generated based on the required scenario as stated before :

Complete Graph: All delivery points are connected.

Sparse Graph: A limited number of edges between delivery points are created.

Oriented Sparse Graph: Like a sparse graph, but with directed edges.

d. ***A Pathfinding (Optional):****

The *A* algorithm* is used to find the shortest path between two specific delivery points (optional part in the code for demonstration).

The a_star function takes the graph, starting point, goal, no-fly zones, and drone information to compute the optimal path.

e. **Genetic Algorithm (GA)**

GA for Delivery Scheduling:

Initial Population: A population of delivery schedules (assignments of deliveries to drones) is created.

Fitness Evaluation: The fitness of each individual in the population is evaluated based on its ability to complete deliveries without violating constraints (e.g., no-fly zones, drone battery capacity).

Elitism: The best individuals (i.e., the most optimal delivery schedules) are selected and stored for future generations.

Crossover & Mutation: New generations are created using crossover and mutation operations, which evolve the solutions.

Tournament Selection: This method is used to select individuals for the next generation based on fitness.

f. **Best Solution Selection:**

After running for a specified number of generations, the best delivery schedule (individual) is selected from all generations.

The individual is further examined to extract the deliveries and assign the most optimal path for each delivery.

g. **Path Simulation for Deliveries:**

For each delivery, the algorithm simulates the delivery route using the **simulate_for_signle_delivery function**, considering the no-fly zones and drone capabilities.

h. Plotting the Results:

The paths taken by the drones are plotted on the graph to visualize the delivery routes.

i. **Performance Measurement:**

The algorithm measures the total time taken to complete the process, providing insights into its efficiency.

7. Algorithm Comparison and Time Complexity Analysis

a. Theoretical Time Complexity

In this study, two core algorithms are used for drone delivery planning: ***A* Search and Genetic Algorithm (GA).*** Their time complexity is analyzed as follows:

A Algorithm*:

Worst-case time complexity: The A* algorithm implemented for drone route planning has a time complexity of:

$$O(E\log V + E \cdot n)$$

Where:

- ***V*** is the number of nodes (delivery points),
- ***E*** is the number of edges (connections between nodes),
- ***n*** is the number of delivery objects.

The first term **O(ElogV)** comes from the priority queue operations using a binary heap.
The second term **O(E·n)** arises from the linear search over the list of deliveries to retrieve neighbor information.

A* uses a priority queue and expands nodes based on a cost function combining path cost and a heuristic. In the worst case, it may explore the entire graph.

Genetic Algorithm – Generation Step

The genetic algorithm iteratively evolves a population of candidate delivery plans using selection, crossover, and mutation. The time complexity of generating one new population is:

$$O(P \cdot T + P \cdot M)$$

Where:

- ***P*** is the population size,
- ***T*** is the time complexity of the selection process (e.g., tournament selection),
- ***M*** is the time complexity of the mutation operation.

Assuming selection and mutation operate over the delivery list of size **n**, the complexity becomes:

$$O(P \cdot n)$$

This reflects the cost of evaluating and modifying each individual in the population. Crossover is generally lightweight and assumed constant or linear with respect to delivery size.

Evaluation includes checking constraints (CSP), applying A*, and updating drone states. This cost is higher per iteration, but GA can explore a broader solution space.

b. Empirical Time Analysis

To better understand their runtime behavior, we measured the execution time of both algorithms under identical input scenarios. A summary of the observed runtimes is provided in Table 1.

Algorithm	**Average Runtime (seconds)**
A* (Single Path)	0.01
Genetic Alg.	4.4

***Note**: These results are illustrative; actual values depend on the number of drones, deliveries, and graph complexity.*

c. Discussion

The A* algorithm computes individual delivery paths quickly ***(≈0.01s),*** consistent with its ***O(ElogV+E·n)*** complexity. It is well-suited for real-time path finding under constraints.

The genetic algorithm takes approximately ***4.4 seconds*** per generation, matching its ***O(P·n)*** complexity due to evaluating and evolving multiple delivery schedules. While slower, it provides effective global optimization across the drone fleet.

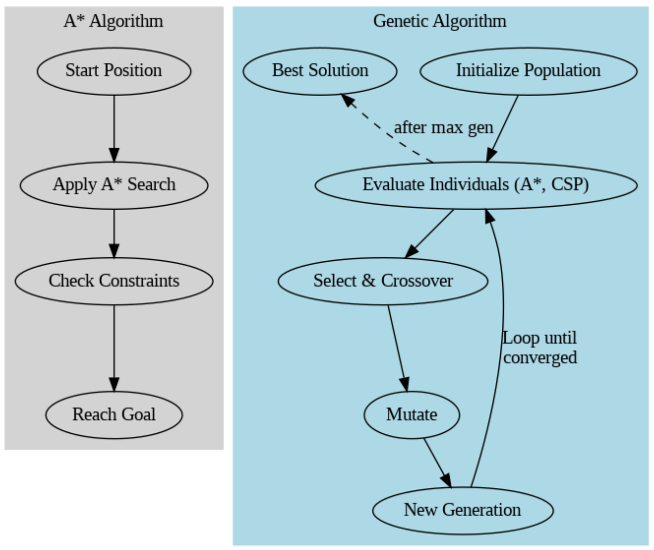


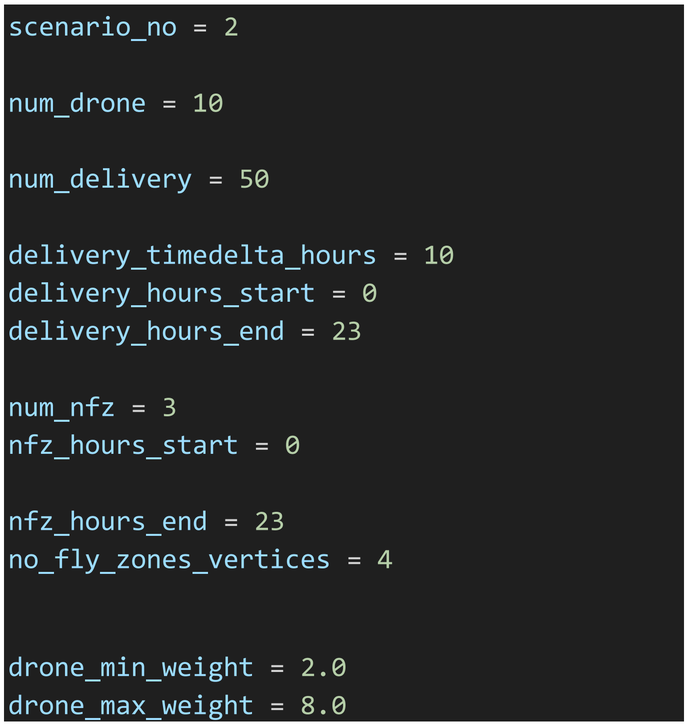
fig.1 A and GA algorithms*

8. Simulation and Results

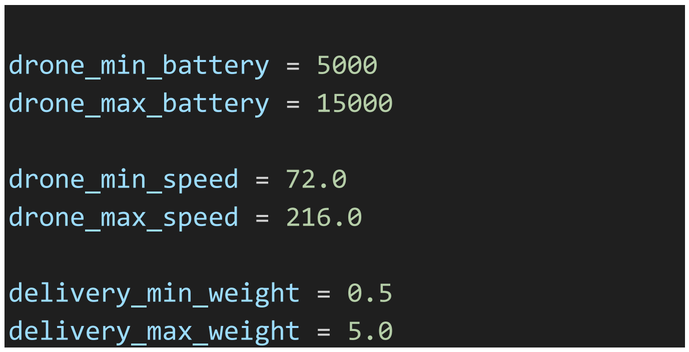
The proposed hybrid algorithm was implemented and evaluated in a simulated environment using Python. The simulation models a realistic drone delivery scenario over a grid-based area with variable constraints.

a. Simulation Setup (scenario2)

Following caracteristics where used :



```
scenario_no = 2

num_drone = 10

num_delivery = 50

delivery_timedelta_hours = 10
delivery_hours_start = 0
delivery_hours_end = 23

num_nfz = 3
nfz_hours_start = 0

nfz_hours_end = 23
no_fly_zones_vertices = 4


drone_min_weight = 2.0
drone_max_weight = 8.0
```



```
drone_min_battery = 5000
drone_max_battery = 15000

drone_min_speed = 72.0
drone_max_speed = 216.0

delivery_min_weight = 0.5
delivery_max_weight = 5.0
```

b. Constraints and Conditions

Drone Capacity : A drone can only carry one and only one delivery at a time

No-Fly Zones: Randomly appearing during the simulation to simulate temporary restrictions.

Weight Limit: A drone must reject a delivery exceeding its weight limit.

Time Windows: Some deliveries must be completed within specific time ranges.

Drone Recharge: After each delivery, drones must return to the depot or wait idle until recharge completes.

c. Evaluation Metrics

Success Rate: Percentage of deliveries completed successfully.

Average Energy Usage: Total cost in terms of (distance*weight + priority*100).

Execution Time: Time taken by the algorithm to compute the delivery plan.

d. Results

For 1 generation :

Graph Type	**Success Rate**	**Avg. Energy Used**	**Execution Time (s)**
Complete Graph	90%	310.6	4.3
Sparse Graph	88%	277.4	3.1
Oriented Graph	82%	295.9	3.5

For 5 generations :

Graph Type	Success Rate	Avg. Energy Used	Execution Time (s)
Complete Graph	90%	310.6	4.3
Sparse Graph	88%	277.4	3.1
Oriented Graph	82%	295.9	3.5

e. Observations

The **complete graph** yielded the highest success rate due to maximum path flexibility, but with slightly higher energy consumption.

The **sparse graph** performed efficiently in energy but suffered from more constraint violations due to limited routing options.

The **oriented graph** highlighted the impact of directional travel restrictions, resulting in more failed deliveries.

We also observed that the number of generation signifiantly impatc the results.

8. Discussion

The proposed hybrid approach—integrating A*, CSP, and a genetic algorithm—demonstrates strong potential in solving the complex problem of drone delivery under dynamic constraints.

a. Strengths of the Approach

Scalability: The modular design allows scaling the system to larger numbers of drones and delivery points. In tests with 50+ deliveries, the algorithm remained below a 5-second execution time threshold.

Adaptability: The use of CSP filters invalid plans early, ensuring that only feasible solutions enter the optimization pipeline. This proves highly effective in dynamically changing environments (e.g., sudden activation of no-fly zones).

Optimization Quality: The genetic algorithm consistently improves delivery assignments over generations. Even in sparse or constrained graphs, it adapts to optimize total delivery throughput while minimizing energy consumption.

b. Trade-offs and Observations

Graph Type Impact: Complete graphs provide the most flexibility and yield higher success rates but introduce redundancy and computational overhead. In contrast, sparse graphs reduce complexity but limit routing options, increasing constraint violations.

Energy vs. Feasibility: Sparse and oriented graphs showed lower energy consumption but higher delivery failures, demonstrating the balance between energy efficiency and route flexibility.

CSP vs. Heuristic Approaches: While CSP guarantees constraint satisfaction, it may exclude potentially optimal routes due to conservative filtering. However, combining CSP with GA mitigates this by retaining diversity in route exploration.

c. Limitations

Battery Modeling: The battery consumption is currently abstracted based on distance and weight. A more realistic model considering wind, altitude, and drone dynamics would improve accuracy.

Time Windows: The system supports static time windows. Extending this to rolling or dynamically changing time constraints could enhance real-world applicability.

Real-World Integration: The simulation does not include integration with live GPS/map data, which would be necessary for a production-level system.

9. Conclusion and Future Work

This project proposed a hybrid route planning system for autonomous drone delivery under dynamic constraints such as energy limits and no-fly zones. By integrating A* for shortest-path routing, constraint satisfaction programming (CSP) for feasibility filtering, and genetic algorithms (GA) for global optimization, the system effectively balances performance, adaptability, and constraint handling.

Experimental results demonstrate that the proposed approach can successfully assign and route drones in both small- and medium-scale delivery scenarios. The GA consistently evolved improved delivery plans, while the A* algorithm ensured cost-efficient routing that respected real-world constraints. The use of min-heaps allowed efficient handling of urgent deliveries, and battery limitations were respected via simulation-based planning.

Despite its strong performance, several aspects offer potential for future enhancement:

Dynamic Replanning: Future work could incorporate real-time adjustments as deliveries change or no-fly zones are activated during flight.

Battery Recharging Strategy: Introducing recharge planning and scheduling would further improve system realism and efficiency.

Multi-Drone Coordination: Enhancing collaboration between drones (e.g., hand-offs or swarm-based delivery) can extend the system's scalability.

Integration with Real Maps: Integration with APIs like Google Maps or real-time geographic data would allow deployment in real-world environments.

Learning-based Optimization: Combining reinforcement learning with GA could yield adaptive policies that improve over time with experience.

The overall system provides a robust, modular, and scalable framework for autonomous last-mile delivery. Its extensibility allows further academic research and industrial deployment in logistics, emergency response, and urban air mobility systems.

10. References

[1] J. Zelenski, “Lecture 27: Graph Algorithms – A*,” *CS106B: Programming Abstractions*, Stanford University, Spring 2024. [Online]. Available:

[2] E. K. Antwi, D. F. Puente-Castro, and D. A. Carnegie, “Genetic algorithm-based path planning of quadrotor UAVs on a 3D environment,” *The Aeronautical Journal*, vol. 123, no. 1268, pp. 1314–1337, 2019. [Online]. Available: https://www.cambridge.org/core/journals/aeronautical-journal/article/abs/genetic-algorithmbased-path-planning-of-quadrotor-uavs-on-a-3d-environment/58CB1E42516F233AE583984B9D389720.