RP2.2
Automated Testing of Unmanned Aerial Vehicles

Author: **Giulio Montuori,** *Professor:* **Matteo Camilli,**

Politecnico di Milano

0 INTRODUCTION

This report presents a UAV test case generator developed for the SBFT-inspired R2.2 project. The tool is designed to find potential vulnerabilities in PX4's obstacle avoidance system by generating test cases using search-based technique. The generator integrates with Aerialist, a test bench built on top of PX4, and uses `NSGA-II` algorithm to explore configurations of static obstacles that might lead to UAV crashes or dangerously close encounters, also known as hard and soft fail in the **UAV Testing Competition**.

1 System Architecture

The system comprises two main components, `start.py` the test generator and `middle.py` the Aerialist docker execution daemon:

- `start.py`: Executes the optimization algorithm (`NSGA-II` via `pymoo`), generates test configurations, and encodes them as `JSON`. This runs on the latest available version of Python.
- `middle.py`: Continuously watches the shared folder, converts the `JSONs` into `YAML` test files, run the simulation via PX4 and Gazebo, and writes back the result in `JSONs`. This run on Pyhton 3.8.0 as required by Aerialist inside the docker container.

Each test cycle involves converting genomes to 3D obstacle placements, running the simulation, extracting flight trajectories, and computing fitness metrics. Inter-process communication is handled via a shared directory.

2 Genetic Encoding and Fitness Objectives

Each test case comprises three non-overlapping box-shaped obstacles, each defined by six genes: `[x, y, l, w, h, r]`. Thus, genomes are 18-dimensional vectors. The following objectives are optimized:

- Minimize distance to obstacles: Simulation returning UAV minimum distance, those under `1,5` are considered risky.
- Maximize diversity: Novelty is quantified via DTW on 2D flight trajectories.
- Minimize flight duration: Shorter test are preferred.

Invalid genomes, like overlapping obstacles, or failed simulations are heavily penalized.

2.1 Trajectory Novelty Metric

To promote behavioral diversity among test cases, the generator computes a novelty score using Dynamic Time Warping (DTW). Each UAV trajectory is down-sampled and transformed into a sequence of `(x, y)` points. The novelty is defined as the DTW distance between the current trajectory and those stored in an archive of previously generated trajectories. Higher DTW values indicate greater novelty.

2.2 Constraints Enforcement

The generator enforces competition constraints:

- Obstacles are non-overlapping.
- All obstacles have height `h > 10`.
- All placed on ground level (`z = 0`).
- Number of obstacles fixed to 3.
- All obstacle parameters respect domain bounds.

The non-overlapping logic is enforced using geometric polygon intersection via `shaply` to ensure physical feasibility of generated obstacle layouts.

3 NSGA-II with Lexicographic Survival

The algorithm uses a custom survival operator `TolerานntLexicoSurvival` to prioritize solutions via lexicographic ordering:

1. Distance: The main source of point.
2. Diversity: To avoid penalty.
3. Time.

Tolerances are applied to avoid over fitting to negligible differences and promote diversity. The initial populations is seeded near the UAV route, via `PathSeedSampling` to improve search convergence, using the way-points listed in the mission plan. A diagnostic logger prints best distances, mean and max novelty and durations score per generation.

4 Simulation Pipeline

Simulations are executed in docker with the `middle.py` daemon that:

- Converts `JSON` test descriptions into `YAML`.
- Constructs `DroneTest` instances and runs them via Aerialist.
- Records trajectories, duration, and minimum distances.
- Applies a timeout fail safe in case of a test structured in a way where the UAV can't reach one of the way-points of the mission plan.

All results are saved as `JSON` files and fetched by the generation once simulations complete. The results includes timestamped trajectory of UAV, which is then parsed to compute the distance and DTW novelty. Simulations that exceed the maximum allowed duration are terminated, and their results are replaced with an error `JSON`. This prevents blocking the optimization loop and ensures robust handling of invalid or stalled tests.

5 Evaluation and Output

Each simulation produces:

- A `YAML` test file, used for the simulation and to build the final folder of top tests.
- A `JSON` result with minimum distance, time, and a timestamped flight path.

At the end of the optimization, the best test cases from all generation are selected using a tolerance filter and saved to a final directory.

6 Reproducibility and Configuration

Parameters such as obstacle bounds, evolution settings, and runtime controls are defined in a centralized file `parameters.py` for ease of tuning and experimentation. The generator ensures termination of docker container using Python's library `atexit` at the and of the execution. The generator is fully deterministic under the same random seed (`SEED=42`). It communicates with the Aerialist daemon via a shared volume between the Docker container and the Linux environment.