SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Effectiveness of Search-Based Testing on a Deep Reinforcement-Learned Swarm Controller

Tianhao Gu

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Effectiveness of Search-Based Testing on a Deep Reinforcement-Learned Swarm Controller

# Effektivität der Suchbasierten Testverfahren auf einem durch Tiefenverstärkung Gelernten Schwarmcontroller

| | |
|---|---|
| Author: | Tianhao Gu |
| Examiner: | Prof. Dr. Alexander Pretschner |
| Supervisor: | David Marson, Simon Speth |
| Submission Date: | 22.09.2024 |

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 22.09.2024 Tianhao Gu

# Acknowledgments

# Abstract

The application of drone swarms in agriculture, military, logistics, public rescue, and other fields has been a growing area of research. Traditionally, these systems rely on a central controller to make decisions for the entire swarm. However, as the number of drones increases, the central controller becomes a bottleneck, leading to delays and reduced efficiency, thus limiting the system's scalability. To overcome this limitation, researchers have recently applied deep reinforcement learning (DRL) to train decentralized controllers of drone swarms , which allows decentralized decision-making and has demonstrated excellent performance in various tasks.

However, before widespread deployment, it is crucial to validate the robustness of these systems. The "black box" nature of DRL controllers makes this validation particularly challenging. To address this issue, we use bio-inspired metaheuristic search, such as genetic algorithm, to perform scenario-based testing on decentralized DRL-based controllers. The approach has been utilized in the autonomous car domain to build relevant scenarios and search for scenario configurations that elicit "edge case" unsafe behaviors of the system under test. Scenario-based testing involves categorizing potential situations into distinct groups, where each group represents a specific type of test scenario. From each group, only a select few representative test cases are chosen. This approach allows for a significant reduction in the total number of tests while still ensuring coverage of the system's critical scenarios. This paper investigates whether scenario-based testing with metaheuristic search is also more effective than random testing on decentralized DRL-based controllers. The evaluation scenario involves a drone swarm navigating from one side of the room to a static goal at the other side in a three-dimensional room populated with multiple static cylinder obstacles. Various fitness functions are at the same time defined to specify safety metrics. For the experiment, a new framework called OpenSBT is used, offering a modular and extensible codebase to facilitate search-based testing. This framework integrates search algorithms, fitness functions, and simulation environments in a modular manner.

# Contents

# 1 Introduction

In recent years, Unmanned Aerial Vehicles (UAVs), commonly referred to as drones, have become a hot topic of research. UAVs have garnered significant attention not only in academic circles, where a growing body of literature explores their various applications and advancements, but also in everyday life, with commercial drones like DJI's consumer-grade models becoming increasingly ubiquitous. The paper [1] provides a comprehensive review of Unmanned Aerial Vehicle (UAV) swarm applications, highlighting their use in entertainment, security, and environmental monitoring. The authors discuss key challenges, such as scalability in real-world environments and limitations in localization and communication, while also proposing a modular system architecture for future development. We focus specifically on decentralized UAV swarms (We use the terms UAV and "drone" interchangeably). The decentralized models offer enhanced scalability, which are crucial for a wide range of real-world applications, from search and rescue missions to environmental monitoring [2].

One approach to implement the decentralized controller is to use DRL, as demonstrated by recent research showing improvements in decision-making accuracy and system stability [3]. These UAVs are interconnected and aware of each other, exchanging information to update their states and avoid collisions while coordinating movements. However, decentralized UAV systems, while scalable, also exhibit several significant drawbacks, such as instability and lack of interpretability.

The instability is the most significant drawback that needs to be considered. During the training of the decentralized controllers, the drones continuously interact with the environment to learn strategies. However, in real-world deployment, controllers with DRL often encounter new environmental factors that were not present during training, leading to unpredictable behavior. For instance, dynamic changes in the environment, perception errors, hardware malfunctions, and external disturbances are often overlooked in training. Additionally, controllers with DRL are often considered as "black boxes", meaning that the internal workings of the model—how it processes input data and makes decisions—are not easily understandable by humans. This lack of transparency is a major flaw in safety-critical applications, as it hampers understanding or predicting why the model makes certain decisions, and it also makes it nearly impossible to analyze and improve the decision making approaching of the controllers. Furthermore, the article [4] explores the explainability challenges with DRL, focusing on

two main approaches: transparent algorithms and post-hoc explainability. It emphasizes the importance of understanding and explaining DRL decisions in critical fields like autonomous driving and healthcare. Despite advancements, there is still no universal method that can be broadly applied across different environments and tasks.s

However, if we want to deploy and apply controllers with DRL in real-world scenarios, we have to verify its robustness and functionality. Since we cannot prove these properties, we need to conduct reliable testing. Due to cost, time and practical constraints, it is not feasible to test all possible cases in a real physical environment, so the simulation is necessary. There are two main challenges in this regard. The first challenge is how to accurately replicate the complex real physical environment in simulations. The second challenge is that, since environmental parameters are not discrete, the combination of all environmental parameters are infinite, making the search space very large. Therefore, how to efficiently find critical test cases in limited time is crucial.

To evaluate the behavior of autonomous systems, current methods apply SBT combined with metaheuristic search, focusing on testing individual UAVs. This approach is adapted from methods used in autonomous vehicles, where it is employed to generate relevant scenarios and identify configurations that provoke "edge case" unsafe behaviors in the System Under Test (SUT).

However, there is currently no existing work that tests the behavior of decentralized drone swarms with DRL. One similar research in terms of DRL is from A. Zolfagharian and M. Abdellatif, etc [5]. It introduces Search-based Testing Approach of Reinforcement Learning Agents (STARLA), a search-based testing method for DRL agents, combining genetic algorithms and machine learning to efficiently identify functional faults. It uses multi-objective fitness functions, including reward, fault probability, and decision certainty, to guide the search toward faulty episodes. The approach reduces the state space through state abstraction and applies crossover and mutation to generate diverse test cases. STARLA is particularly useful for detecting faults in safety-critical applications under limited testing budgets, focusing on realistic and critical areas of DRL policies.

In the following chapters, We will first explore the classification, characteristics and applications of UAVs. Following this, we will introduce the foundations of SBT with metaheuristic search, the functionality of OpenSBT framework. Then we will discuss the experimental environment, including the training of various controllers, highlighting their differences and training environments. In the next section, we will conduct the experiments, collect and then analyze the data, especially the changes of fitness function values based on different simulation variables and generation numbers. Finally, we will analyze the shortcomings of the experimental methods, summarize the key findings, and provide suggestions and directions for future research.

# 2 Experimental Background

## 2.1 Unmanned Aerial Vehicles

### 2.1.1 Taxonomy of Multi-Robot Systems

This paper [6] proposes a classification framework for coordination in Multi-Robot Systems (MRS) , exploring a hierarchical mechanism from cooperation, knowledge, and coordination to organization. It is particularly applicable to complex tasks in dynamic environments. The study highlights the effectiveness of strong coordination systems in handling complex tasks and the flexibility and robustness of distributed systems in dynamic settings. The figure below illustrate the hierarchical structure of MRS.

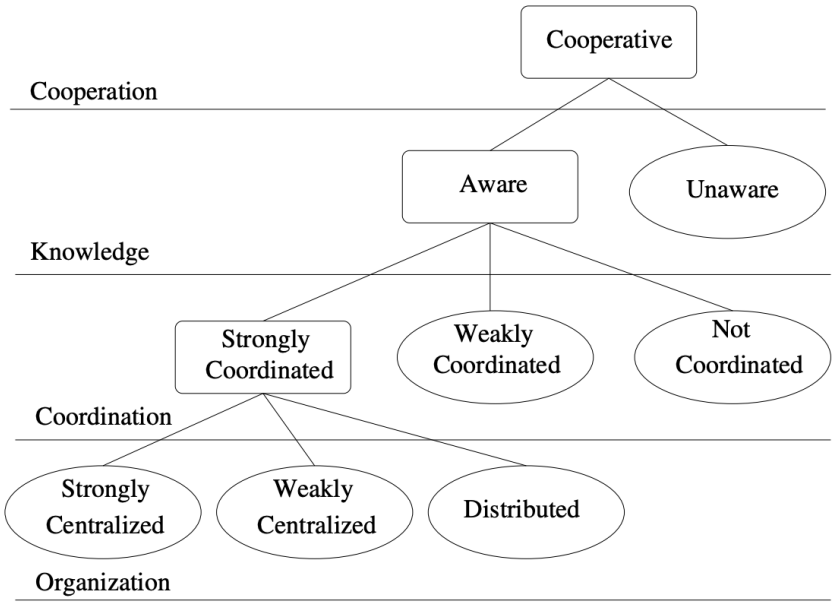
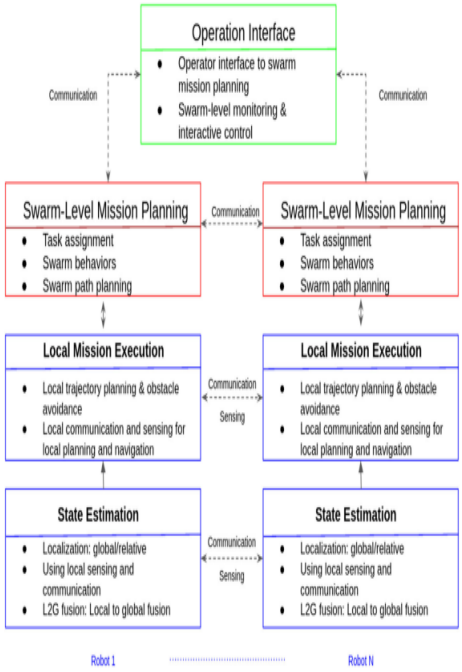


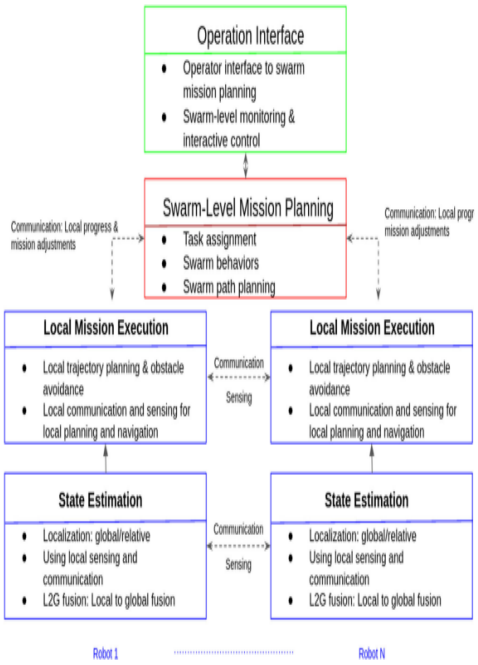

Figure 2.1: MRS Taxonomy

### 2.1.2 Comparison of Centralized and Decentralized Architectures

In swarm robotic mission planning, centralized and decentralized architectures differ in task management. The centralized approach relies on a unified "Operation Interface" for overseeing mission planning and control, with tasks assigned through the "Swarm-Level Mission Planning" module. Each robot executes local tasks based on this centralized plan, using local sensing and communication for state estimation and obstacle avoidance. In contrast, the distributed architecture gives individual robots autonomy in task assignment and path planning. Each robot independently manages its mission, coordinating through communication and sensing, leading to greater flexibility and robustness in completing tasks [1].




(a) `Centralized Mission Planning Architecture`




(b) `Decentralized Mission Planning Architecture`

Table 2.1: Comparison of Swarm Robotic Mission Planning Architectures

### 2.1.3 Application Fields of Drone Swarms

Drone swarms offer versatility across various fields due to their advanced capabilities. In entertainment, they are used for synchronized light shows and aerial performances. For security and surveillance, drone swarms enhance monitoring of large areas through

RGB and thermal cameras, providing real-time data. In collaborative transportation, multiple drones can work together to carry heavier loads, though coordination remains a challenge. Additionally, for environmental monitoring, equipped with sensors, they quickly cover vast areas to collect data on air quality, forest fires, and pollution.

This interesting study [7] examines the advancements in Swarm UAVs, emphasizing localization challenges and techniques for autonomous UAV operations. It explores methods like computer vision, cooperative localization, and machine learning, focusing on improving accuracy and reliability in dynamic environments such as agriculture and disaster relief. Future directions include integrating 5G, energy harvesting, and enhanced security. While providing a comprehensive overview, the study highlights the need for practical solutions to overcome scalability and energy limitations.

### 2.1.4 Features of Quadrotor Swarms with End-to-End DRL

Quadrotor swarms trained with end-to-end deep reinforcement learning display several key features [3]:

- **Adaptability of the Model:** The approach does not rely on predefined physical models, allowing for flexibility when applied to quadrotors with varying attributes such as mass, size, inertia, and thrust. This adaptability is achieved by retraining the model within a modified simulation environment.
- **Efficient Transfer and Scaling:** The trained policies can be smoothly implemented on real-world systems to perform tasks effectively. Moreover, with additional training, the model can handle up to 128 quadrotors, maintaining computational efficiency with only a minor increase in resource demands per drone.

### 2.1.5 Controller Models of Quadrotor Swarms with End-to-End DRL

The end-to-end deep reinforcement learning models designed for decentralized quadrotor swarm control utilize two primary neural network frameworks [3]:

- **Deep Sets Architecture:** This framework utilizes a deep sets encoder to create representations of each quadrotor's surrounding environment that remain unaffected by permutations and scaling. The encoder processes the observed data from nearby quadrotors and aggregates the information using a permutation-invariant function. However, this method does not differentiate the varying importance of different neighboring quadrotors.
- **Attention-based Architecture:** This approach integrates an attention mechanism to evaluate the relevance of each neighboring quadrotor based on their state

and relative position. The encoder generates attention scores to prioritize neighbors that are closer and moving more quickly, which improves performance in trajectory planning and decision-making.

## 2.2 Foundations of SBT with Metaheuristic Search

### 2.2.1 Concept of SBT with Mataheuristic Search

SBT is a widely used method to evaluate the performance and functionality of systems, particularly in autonomous vehicles, by simulating real-world scenarios. In SBT, the SUT operates as a black box, focusing on how the system performs under realistic or hypothetical conditions to identify potential issues. In the field of autonomous driving, scenario-based testing is more efficient than distance-based validation methods. By systematically generating, adjusting, and validating various driving scenarios, scenario testing effectively captures potential issues under real or hypothetical conditions. Compared to validation methods that rely on driving distance, scenario testing not only reduces verification costs but also ensures broader coverage of safety scenarios. This approach is particularly advantageous for testing highly automated vehicles, where comprehensive and cost-effective validation is crucial [8].

Metaheuristic search is an advanced optimization technique that efficiently solves complex problems using algorithms like Genetic Algorithm (GA) and Particle Swarm Optimization (PSO) [9], making it ideal for economic and management decision-making applications. We combine SBT with metaheuristic search and relevant studies have found that heuristic search techniques have successfully optimized scenario testing for automated vehicles, with focus on automated driving [10]–[13].

### 2.2.2 Abstraction Levels of Scenarios

The abstraction levels of scenarios are crucial for understanding and testing a system's behavior. There are different ways to define scenarios and in this paper we use functional scenarios, logical scenarios, and concrete scenarios to define the scenario [14]:

- **Functional Scenarios:** Described using natural language, these scenarios focus on the system's functionality and user interactions, without involving specific numerical values or details.
- **Logical Scenarios:** These scenarios provide an abstract view of the system's behavior by defining parameter ranges. They describe the behavior in terms of ranges of values rather than specific numbers.

- **Concrete Scenarios:** Concrete scenarios are detailed instances of logical scenarios, including exact parameter values and actions.

### 2.2.3 Metaheuristic Search Algorithms

#### Genetic Algorithm

GA is a type of metaheuristic algorithm that leverage principles from natural selection and genetics, imitating biological evolution to discover optimal solutions [15]. Non-dominated Sorting Genetic Algorithm II (NSGA-II) is a specific example which widely applied in multi-objective optimization problems. As shown in *Figure 2.2*, the primary steps involved in genetic search in NSGA-II include selection, crossover, mutation, and iteration:

1. **Initial Population Creation**: A random population is generated, with each individual representing a potential solution.
2. **Fitness Evaluation**: Every individual is assessed based on a predefined fitness function to determine their fitness value.
3. **Selection**: Individuals are chosen according to their fitness. NSGA-II employs a selection process that utilizes fast non-dominated sorting and crowding distance, giving preference to individuals with lower ranks and those with higher crowding distances within the same rank to ensure diversity in the population.
4. **Crossover**: Chosen individuals undergo crossover to generate new offspring, helping prevent the search from getting stuck in local optima. Crossover mimics genetic recombination by exchanging sections of the genes from parent individuals to produce varied offspring.
5. **Mutation**: Offspring are subjected to mutation, simulating random genetic changes to enhance population diversity.
6. **Iteration**: The steps of fitness evaluation, selection, crossover, and mutation are repeated until either a set number of iterations is reached or the convergence criteria are satisfied.

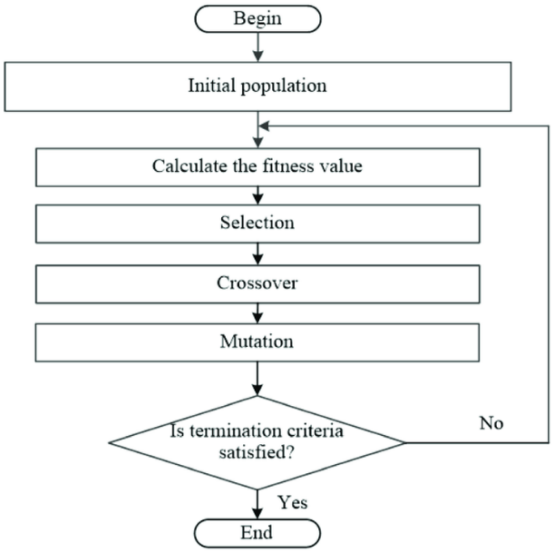



Figure 2.2: Flowchart of the Standard Genetic Algorithm

**Particle Swarm Optimization and Its Variants**

PSO is an optimization algorithm inspired by the foraging behavior of birds, and it has evolved into several variants to enhance its performance across diverse optimization problems in recent research . One notable variant is Particle Swarm Optimization with Increasing Topology Connectivity (PSO-ITC), which gradually increases the connectivity between particles to balance global and local search capabilities. This approach has demonstrated superior performance in solving unconstrained single-objective optimization problems and engineering design challenges. PSO-ITC has shown improvements in convergence speed and solution accuracy compared to other variants of PSO [16]. Another simplified PSO variant reduces the complexity of the algorithm while maintaining comparable or slightly better performance in optimizing neural network problems [17]. PSO variants continue to evolve, enhancing optimization efficiency in various engineering and real-world applications. The flowchart of basic PSO is show in *Figure 2.3*.

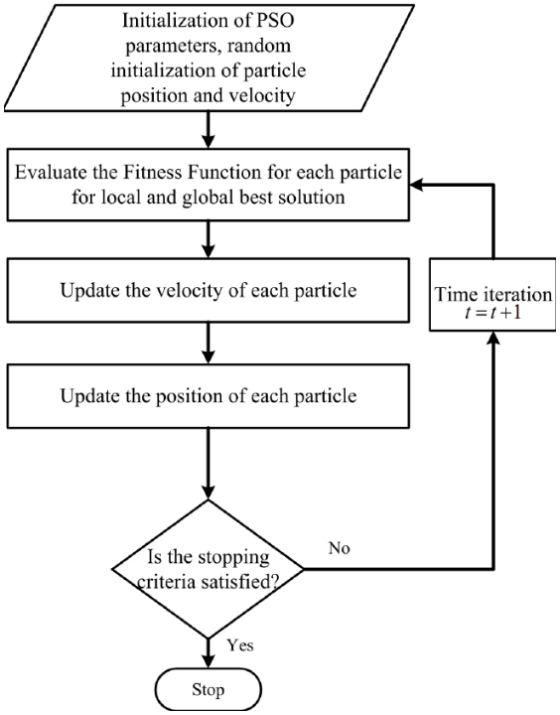



Figure 2.3: Flowchart for Basic PSO Algorithm

## 2.3 OpenSBT Framework

### 2.3.1 The Purpose of the OpenSBT Framework

OpenSBT is a new framework developed by Fortiss and the purpose of the OpenSBT framework is to facilitate search-based testing by providing a modular and extendable code base. This framework integrates various search algorithms, fitness/critical functions, and simulation environments in a cohesive manner. OpenSBT offers interfaces for metaheuristic search algorithms, fitness evaluation to create a comprehensive testing environment. The framework supports the extension of internal optimization models, such as Problem and Result, to tailor heuristic search algorithms specifically for our use case, thus enhancing the effectiveness and efficiency of the testing process [18].

### 2.3.2 The Architecture of the OpenSBT Framework

*Figure 2.4* illustrates the architecture of OpenSBT. First, the parameterized scenario generates different test case instances through the metaheuristic search module, where

each set of parameters corresponds to a scenario instance. These instances are passed to the simulator for execution. The output is then processed by the evaluation module, which assesses the performance of the test cases using the fitness function interface. The optimizer interface works in conjunction with the heuristic search to adjust the parameters, generating improved scenarios. Now, we will explain each component in detail:

- **The Scenarios:** A scenario consists of a series of scenes, where each scene is a snapshot of the environment, encompassing all static and dynamic elements, participating actors, observers, and the relationships among these entities. In OpenSBT, a scenario can be an OpenSCENARIO file or another format supported by the simulator. Alternatively, scenarios can be integrated within the simulator module itself, eliminating the need for a separate scenario file.
- **The Simulation Variables:** Simulation variables are parameters that are modified during the experiment to create various test scenarios. These variables may include initial speed, position, direction, and other specific parameters. By adjusting the values of search variables, diverse test scenarios can be generated.
- **The Search Space:** The search space is defined by arrays of upper and lower bounds of the simulation variables, which establish the valid range for the simulation variables. These bounds ensure that generated scenarios remain within reasonable and controllable limits.
- **The Fitness Function:** The fitness function evaluates the performance of each test case, assessing how well it meets the expected test goals and requirements.
- **The Critical Function:** The criticality function assesses safety requirements or test criteria to evaluate the safety and criticality of a test case. These functions may be based on regulatory standards, safety distances, collision probabilities etc.. This function is optional.
- **The Simulation Function:** The simulation function triggers the simulator to execute the test scenarios. It initiates the simulator and manages the execution flow of the experiment, ensuring that the simulation proceeds as planned. This is the most complex and crucial function in the OpenSBT framework. It is important to note that, at present, OpenSBT does not support standalone SUT and scenarios as different modules. Therefore, the SUT must be integrated within the simulation function.

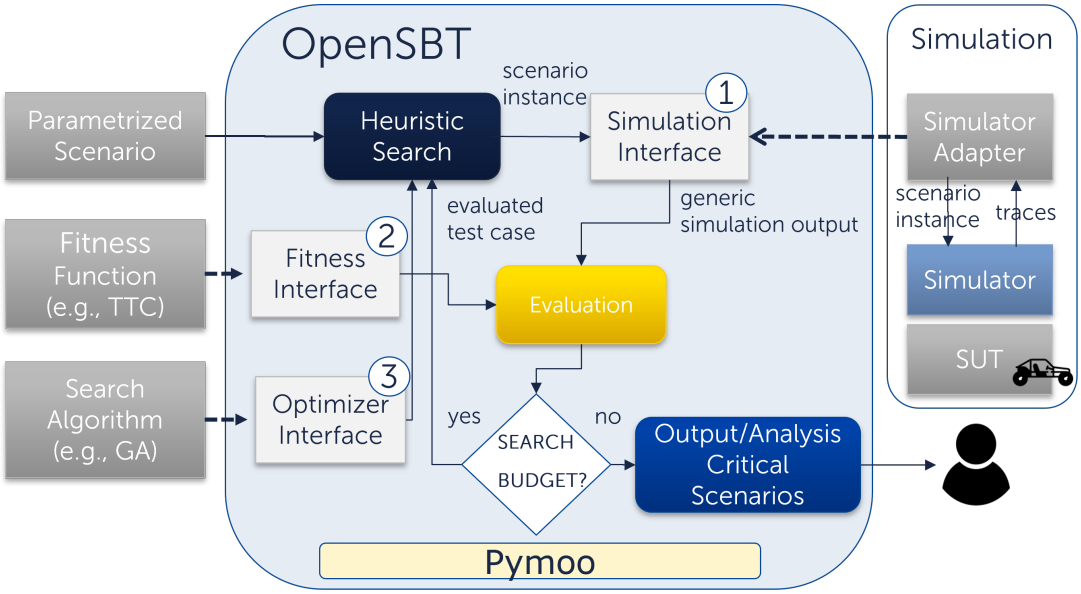



Figure 2.4: The Architecture of OpenSBT

# 3 Experimental Environment

## 3.1 Experimental Objectives

The experiments aim to demonstrate the *effectiveness* of SBT with metaheuristic search when applied to decentralized quadrotor swarms with DRL.

***Effectiveness*:**
The *effectiveness* of SBT with metaheuristic search is measured relative to random testing. Specifically, it is evaluated by that with the same simulation episodes whether SBT with metaheuristic search can identify more critical test cases than using normal random testing.

## 3.2 System Under Test

For our SUT, we reference the work of Sumeet Batra, Zhehui Huang, Aleksei Petrenko, and others, which successfully applied end-to-end DRL to the decentralized controller of drone swarms [3]. Using this research as a foundation, we independently trained two models tailored to the needs of our experiments by setting up proper training parameters. A thorough understanding of the SUT will facilitate analysis and interpretation of our experimental results. In this section, we will briefly discuss the training configurations and the behavioral difference between the two controllers.

### 3.2.1 Training Configurations of Controller1 and Controller2

In this section, we present the training configurations of two controllers, **Controller1** and **Controller2**, used in the context of multi-quadrotor control environments. Both controllers employ DRL techniques to solve complex navigation and obstacles avoidance tasks with multiple quadrotor agents. The key parameters of both controllers are detailed below, with differences highlighted to reflect how variations in settings impact performance, learning dynamics, and policy optimization.

I conducted the training on a cloud server equipped with an RTX 4080Ti GPU and 256 CPU cores, where the level of CPU parallelism was crucial for the total duration of training. The training configuration [3] utilized 1600 parallel processes to collect experience and assigned one GPU per trained policy for inference and

backpropagation. The total training time was approximately six hours. According to prior research we referenced, each training session was equivalent to roughly 15 months of quadrotor simulation. This underscores the importance of fast simulated environments, as collecting an equivalent amount of experience in the real world would be prohibitively expensive [3].

### Controller1: Training Configuration

The training configuration of **Controller1** is as follows:

```
--env=quadrotor_multi --train_for_env_steps=1000000000 --algo=APPO --
    ↪use_rnn=False
--num_workers=200 --num_envs_per_worker=8 --learning_rate=0.0001 --
    ↪ppo_clip_value=5.0
--recurrence=1 --nonlinearity=tanh --actor_critic_share_weights=False
--policy_initialization=xavier_uniform --adaptive_stddev=False --
    ↪with_vtrace=False
--max_policy_lag=100000000 --rnn_size=256 --gae_lambda=1.00 --
    ↪max_grad_norm=5.0
--exploration_loss_coeff=0.0 --rollout=128 --batch_size=1024 --with_pbt=
    ↪False
--normalize_input=False --normalize_returns=False --reward_clip=10 --
    ↪quads_num_agents=8
--quads_obs_repr=xyz_vxyz_R_omega_floor --quads_episode_duration=15.0
--quads_encoder_type=attention --quads_neighbor_visible_num=-1 --
    ↪quads_neighbor_obs_type=pos_vel
--quads_neighbor_hidden_size=256 --quads_neighbor_encoder_type=attention
--quads_collision_reward=5.0 --quads_collision_hitbox_radius=2.0
--quads_collision_falloff_radius=4.0 --quads_collision_smooth_max_penalty
    ↪=10.0
--quads_use_obstacles=True --quads_obst_spawn_area 8 8 --
    ↪quads_obst_collision_reward=5.0
--quads_obstacle_obs_type=octomap --quads_use_downwash=True --
    ↪quads_domain_random=True
--quads_obst_density_random=True --quads_obst_density_min=0.03 --
    ↪quads_obst_density_max=0.15
--quads_obst_size_random=True --quads_obst_size_min=0.3 --
    ↪quads_obst_size_max=3.5
--quads_obst_hidden_size=256 --quads_obst_encoder_type=mlp --
    ↪quads_obst_collision_reward=5.0
```

```
--quads_use_downwash=True --quads_use_numba=True --quads_mode=mix
--anneal_collision_steps=300000000 --replay_buffer_sample_prob=0.75
--save_milestones_sec=3600
```

**Controller1** utilizes a comprehensive setup that integrates eight agents, a dense obstacle environment, and specific attention-based encoders for both agents and obstacles. This controller is suited for complex quadrotor interactions where collisions, multi-agent coordination, and dynamic obstacle avoidance play significant roles. The use of attention mechanisms for both neighbor and obstacle encoding allows the controller to better capture the relevance of the surroundings in decision-making, a critical aspect in decentralized control settings. Several key features such as attention-based encoding (`-quads_encoder_type=attention`), obstacle size and density variations (`-quads_obst_density_random=True`, `-quads_obst_size_random=True`), and collision penalties (`-quads_collision_reward=5.0`) make the controller capable of handling challenging coordination and navigation tasks while avoiding obstacles.

### Controller2: Training Configuration

The training configuration of **Controller2** is as follows:

```
--env=quadrotor_multi --train_for_env_steps=1000000000 --algo=APPO --
    ↪use_rnn=False
--num_workers=200 --num_envs_per_worker=8 --learning_rate=0.0001 --
    ↪ppo_clip_value=5.0
--recurrence=1 --nonlinearity=tanh --actor_critic_share_weights=False --
    ↪policy_initialization=xavier_uniform
--adaptive_stddev=False --with_vtrace=False --max_policy_lag=100000000 --
    ↪rnn_size=256 --gae_lambda=1.00
--max_grad_norm=5.0 --exploration_loss_coeff=0.0 --rollout=128 --
    ↪batch_size=1024 --with_pbt=False
--normalize_input=False --normalize_returns=False --reward_clip=10 --
    ↪quads_use_numba=True
--save_milestones_sec=3600 --anneal_collision_steps=300000000 --
    ↪replay_buffer_sample_prob=0.75
--quads_mode=mix --quads_episode_duration=15.0 --quads_obs_repr=
    ↪xyz_vxyz_R_omega_floor
--quads_neighbor_hidden_size=256 --quads_neighbor_obs_type=pos_vel --
    ↪quads_collision_hitbox_radius=2.0
--quads_collision_falloff_radius=4.0 --quads_collision_reward=5.0 --
    ↪quads_collision_smooth_max_penalty=4.0
```

```
--quads_neighbor_encoder_type=no_encoder --quads_neighbor_visible_num=6 --
    ↪quads_encoder_type=attention
--quads_use_obstacles=True --quads_obst_spawn_area 8 8 --
    ↪quads_obst_collision_reward=5.0
--quads_obstacle_obs_type=octomap --quads_use_downwash=True --
    ↪quads_domain_random=True
--quads_obst_density_random=True --quads_obst_density_min=0.05 --
    ↪quads_obst_density_max=0.2
--quads_obst_size_random=True --quads_obst_size_min=0.3 --
    ↪quads_obst_size_max=0.6
```

**Controller2** employs a similar setup to **Controller1**, but with some critical differences that simplify certain aspects of the environment and agent interaction. Notably, `Controller2` uses fewer visible neighbors (`-quads_neighbor_visible_num=6`) and does not encode neighbors with attention, opting instead for a simpler encoder (`-quads_neighbor_encoder_type=no_encoder`). Additionally, the smooth maximum penalty for collisions is reduced (`-quads_collision_smooth_max_penalty=4.0`), making collisions less punitive compared to **Controller1**. In the subsequent experimental section, it becomes evident that **Controller2** demonstrates significantly weaker obstacle avoidance capabilities compared to **Controller1**. Moreover, It is important to note that the research we are referencing demonstrates strong capabilities in maintaining formation, keeping drones at appropriate distances, and approaching the goal efficiently. However, it only possesses rudimentary obstacle avoidance abilities. The authors post in [19] that research and development are still ongoing to reduce the number of collisions in multi-drone scenarios. *Table 3.1* shows an example episode in DRL training process.

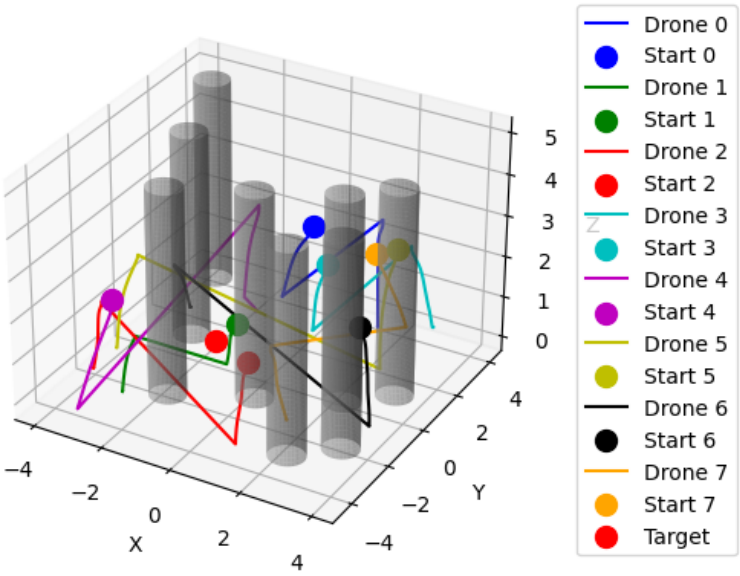
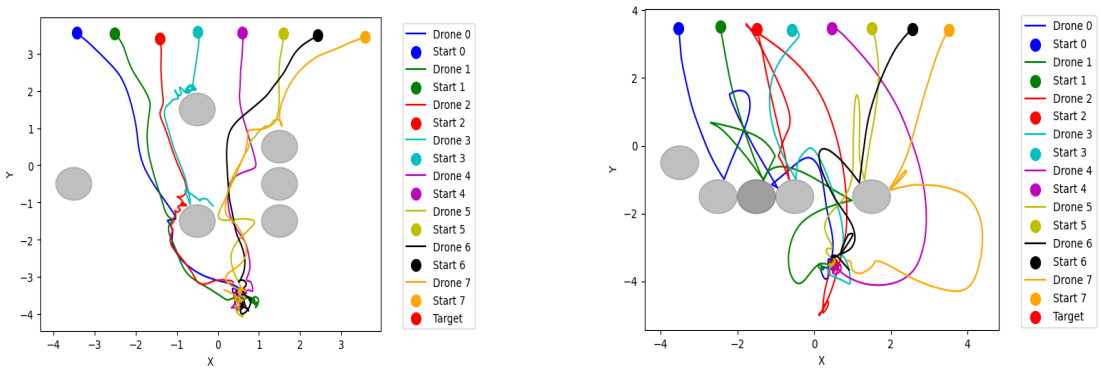



Table 3.1: An Example Episode in DRL-Training

### 3.2.2 Comparison of the Behaviors of Controller1 and Controller2

To provide a more intuitive comparison between **Controller1** and **Controller2**, *Table 3.2* illustrates the visualization of the test case scenarios for both controllers. For clearer observation, a 2D top-down perspective has been employed. The visualizations depict eight drones attempting to navigate from one side of the room with multiple cylindrical obstacles to the goal. A detailed explanation of the scenarios and tasks will be provided in subsequent sections. From *Table 3.2*, it is evident that under the control of **Controller1**, the drones tend to halt when approaching obstacles and adjust their movement direction accordingly. In contrast, the drones controlled by **Controller2** frequently collide directly with the cylindrical obstacles and bounce off without making necessary adjustments.




(a) `A Test Case with Controller1` (b) `A Test Case with Controller2`

Table 3.2: Comparison of the Behaviors of Controller1 and Controller2

## 3.3 Scenarios with Multi-Quadrotor Agents

### 3.3.1 Functional Scenarios

Eight quadrotors, generated on one side of a room, need to avoid multiple fixed cylindrical obstacles along the way and prevent collisions with neighboring drones, obstacles, walls, the floor and ceiling, while approaching the goal on the other side of the room. After reaching the goal, they must maintain a fixed formation upon stabilization.

In real-world deployments, the ability of quadrotors to accomplish similar tasks is of significant practical value. For example, in disaster relief operations, UAVs can work together to search for survivors, assess structural damage, and deliver emergency supplies in areas affected by earthquakes or wildfires. These UAVs must be capable of traversing complex environments, avoiding obstacles, and preventing inter-vehicle collisions while efficiently locating their targets.

### 3.3.2 Logical Scenarios

For a × b × c, a, b, and c represent the length, width, and height, respectively. *Table 3.3* provides a clear overview, facilitating the understanding of the coordinate representation.

**Room Size:** 10 × 10 × 10 meters
**Spawning Area of Obstacles:** 8 × 4 × 10 meters area in the middle of the room, namely the green area in *Table 3.3*. Ignoring the height dimension, the spawning area is divided into 1 × 1 meters squares, each identified by coordinates (x, y), where (0, 0) is the central point of the room. Each square can be an initial position for at most one obstacle and we use the coordination of the central point to represent each square. What needs to be noted here is that the height of the obstacle is fixed as 10 meters. Since the drone cannot pass over or under the obstacles, the vertical coordinate can be often ignored. The rationale for dividing the space into different squares lies in the fact that part of the simulator we are using is derived from the research we referenced, which imposes constraints on how obstacles are generated [3].
**Radius of the Cylindrical Obstacles:** from 0.75 to 1.5 meters
**Fixed Drones Initial Positions:** The blue area in *Table 3.3* and points with different color represent different drones. Each square is an initial position for one drone. The height of all initial positions is fixed as 2 meters.
**Fixed Goal Position:** The red point in *Table 3.3*, namely the square with central position of (0.5, -3.5). The height is fixed as 2.25 meters

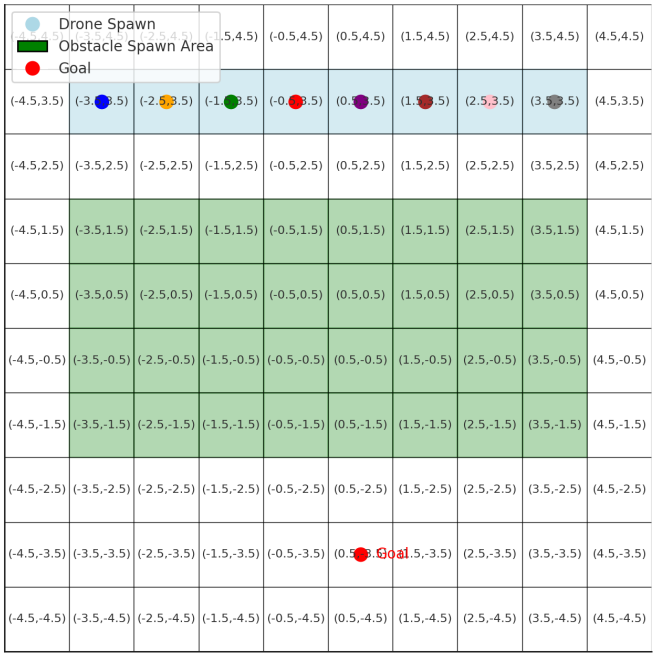



Table 3.3: Overview of Coordinate Representation

### 3.3.3 Concrete Scenarios

Following is one concrete test scenario and its top-down visualization:

| |
|---|
| Eight quadrotors, generated at squares (-3.5, 3.5, 2.), (-2.5, 3.5, 2.), (-1.5, 3.5, 2.), (-0.5, 3.5. 2.), (0.5, 3.5, 2.), (1.5, 3.5, 2.), (2.5, 3.5, 2.), (3.5, 3.5, 2.), respectively, navigate towards the goal at square (0.5, -3.5, 2.25). The quadrotors try to avoid collisions with neighboring drones, walls, the floor, the ceiling and six fixed cylindrical obstacles at (-3.5, 1.5), (-2.5, 1.5), (-3.5, 0.5), (1.5, -0.5), (2,5, 1,5), (3.5, 0.5), each with a radius of 0.97. After reaching the goal, they must maintain a fixed formation upon stabilization. |

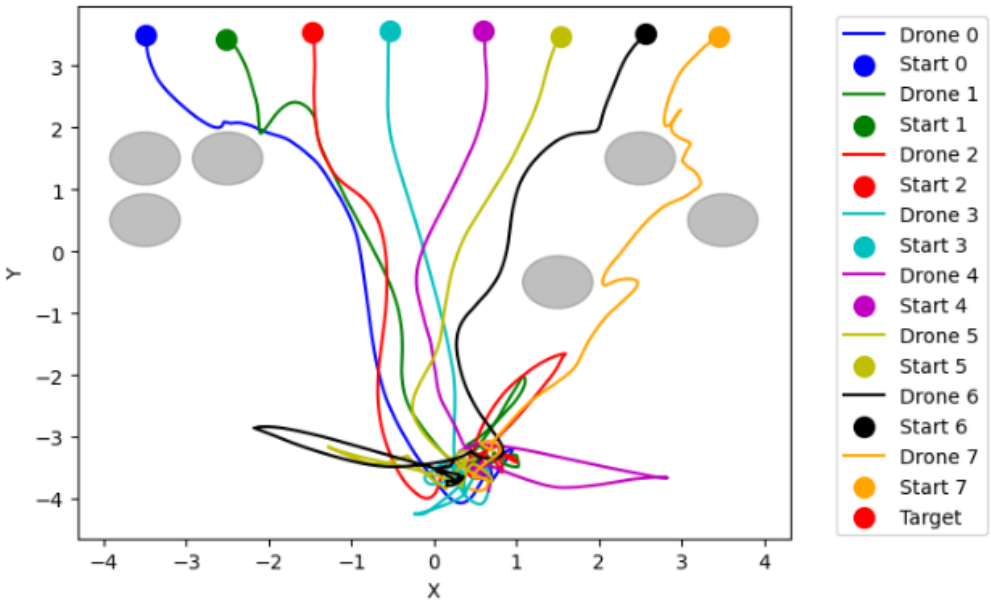



Table 3.4: An Example Test Case

## 3.4 Simulation Variables

In our testing simulation, we define a set of variables and corresponding bounds that govern the configuration of the environment. As the core objective of this study is to effectively identify critical test cases by altering the characteristics of obstacles. Therefore, we have only selected the parameters that are obstacles related such as the

positions and size of the obstacles. Additionally, we have not included the number of obstacles as a simulation variable because the probability of safety issues increases as the number of obstacles increases. This is a self-evident conclusion that does not require experimental validation. However, the number of collisions does not always increases with the size of the obstacles, as it also depends on their placement. In some cases, smaller obstacles can create a "narrow passage," significantly raising the likelihood of collisions as the drone attempts to navigate through it. Second, when selecting simulation variables, we must strike a balance between the number of variables and the controllability of the search space. Too many simulation variables can lead to an exponential expansion of the search space, significantly reducing the efficiency of the search algorithm or even causing it to fail [20]. In such cases, the algorithm may be unable to find critical test cases within a reasonable time frame. On the other hand, too few simulation variables also pose challenges, as critical test cases may only emerge from specific combinations of variables. For instance, when obstacles "surround" the goal, such a combination may create a critical test case [21]. Therefore, when selecting simulation variables, we must consider both the diversity of the variables and the size of the search space to ensure that critical test cases can be identified efficiently and accurately. Moreover, if the search space required in future experiments becomes too large, leading to reduced algorithm efficiency, we can address this issue by replacing *FloatRandomSampling* with *IntegerRandomSampling* in the optimizer.and redefining the simulation variables as an integer list. The final simulation variables for the single-objective experiment in the following chapter include the obstacle size and the positions of multiple obstacles, as listed below:

- `quads_obst_size`: the size of the obstacles, bounded between 0.75 and 1.5. We choose the same range of this parameter as during the DRL training process.
- `quads_obst1_position` to `quads_obst6_position`: these variables define the positions of six obstacles within the range [1.6, 4.799]. The obstacle positions are scaled by a factor of 10, and then rounded to integers for grid placement. Specifically, obstacle positions are sequentially from 0 to 63, corresponding to the squares (-3.5, 3.5) to (3.5, -3.5) in *Table 3.3*.

## 3.5 Fitness Functions

The fitness function is a critical component in our experiment as it is directly linked to the evaluation of safety metrics. Specifically, the fitness functions described here pertains to the testing phase, although in DRL (Deep Reinforcement Learning) process, fitness functions are also used to quantify the model's performance in achieving specific

objectives. A fitness function may consist of multiple sub-functions, each evaluating different aspects of the model's behavior, and it can return a tuple of values representing various objectives. This multi-objective optimization approach is commonly employed when balancing competing goals, such as maximizing performance while minimizing risk. In our experiment, we focus on a single safety metric at a time, where each fitness function that quantifies the model's adherence to a specific safety criterion. Across all experiments, we define that a lower fitness function value indicates a more critical and dangerous test case.

### 3.5.1 Number of Collisions with Obstacles

This fitness function quantifies the number of collisions between the drones and the obstacles. We take the negative value as penalty.

$$\boxed{f_{\text{collisions}}(C_{ij}) = -\sum_{\{i,j\}} C_{ij}} \tag{3.1}$$

where:

- $C_{ij}$ represents the number of collisions between drone $i$ and obstacle $j$,
- The sum is taken over all pairs of drones and obstacles.

### 3.5.2 Average Distance to Obstacles

This fitness function calculates the average minimum distance between the drone on the far left (drone 0) and each obstacles during the flight. It serves as a measure of how safely the specific drone navigates around obstacles. This function can be adjusted to apply for other drones.

$$\boxed{d_{\text{avg}}(0) = \frac{1}{n}\sum_{j=1}^{n} \min_{t} \left( \|p_0(t) - p_j(t)\| \right)} \tag{3.2}$$

where:

- $p_0(t)$ is the position of drone 0 at time $t$,
- $p_j(t)$ is the position of obstacle $j$ at time $t$,
- $n$ is the total number of obstacles.

### 3.5.3 Minimum Distance to Ceiling

This fitness function measures how closely a drone approaches the ceiling, aiming to ensure that drones maintain a safe distance from the ceiling during flight. Similar funtions can be defined for the walls and the floor.

$$\boxed{d_{\text{min\_ceiling}} = \min_{i,t} \left(\text{CeilingHeight} - z_i(t)\right)} \tag{3.3}$$

where:

- $z_i(t)$ is the height (z-coordinate) of drone $i$ at time $t$,
- CeilingHeight is the constant height of the ceiling.

### 3.5.4 Maximum Distance to Goal After Settling

This fitness function evaluates the maximum distance of any drone from the goal after the drone swarms have settled. It ensures that, once the drone swarm has settled, all drones maintain a stable distance from the goal. If the max in the equation is replaced with min, it measures whether all drones can maintain a safe distance from the goal after the swarm has settled,

$$\boxed{d_{\text{max\_goal}} = -\max_{i,t} \left(\|p_i(t) - g\|\right)} \tag{3.4}$$

where:

- $p_i(t)$ is the position of drone $i$ at time $t$,
- $g$ is the fixed goal position.

### 3.5.5 Maximum Distance Between Agents After Settling

This fitness function calculates the maximum distance between any pair of drones after the drone swarms have settled. It ensures that the drones maintain a stable distance from one another. If the max in the equation is replaced with min, it measures whether all drones can maintain a safe distance from each other after the swarm has settled,

$$\boxed{d_{\text{max\_inter\_drone}} = -\max_{\{i,j\}, i \neq j} \left(\max_t \|p_i(t) - p_j(t)\|\right)} \tag{3.5}$$

where:

- $p_i(t)$ is the position of drone $i$ at time $t$,
- $p_j(t)$ is the position of drone $j$ at time $t$.

As mentioned earlier, there are fitness functions involved in the DRL process, and a key issue to consider is whether the same evaluation metrics used during the training phase should also be applied in the testing phase. Training metrics are designed to reflect specific aspects of the model's performance during training and are adjusted according to the final application scenario. For example, in DRL, penalties are applied to the model's collision behavior, as discussed in the training configuration of our controllers. As training progresses, the probability of collisions decreases. Thus, metrics like the reduction in collision frequency are closely tied to the reward and penalty mechanisms in DRL. This provides a strong rationale for believing that using the same metrics to evaluate the model's final performance is often feasible. However, to prevent the model from overfitting during training, auxiliary functions or penalty mechanisms that should not appear in the final fitness function during testing phase are sometimes introduced. For instance, when training a self-driving car model, an additional penalty might be applied if the vehicle maintains the same speed for an extended period. This penalty is meant to prevent the model from becoming too adapted to fixed-speed driving in the training data and thus unable to handle varying traffic conditions. However, in the actual testing and final evaluation phases, this penalty is no longer necessary, as real-world vehicles are not subject to such constraints [22].

Moreover, the provided fitness functions offer a comprehensive evaluation of the safety metrics within the scenario. However, in the subsequent experimental section, we will not be utilizing all of these fitness functions. This is because some of them have a weak correlation with our simulation variables. For instance, the position and size of obstacles have little relevance to the state of the drone swarms after they have settled, rendering the use of certain search algorithms meaningless.

## 3.6 Metaheuristic Search Configuration

We first use the `DefaultSearchConfiguration` class in OpenSBT to initialize the search algorithm's parameters. The key parameters are:

- `n_generations`: It defines the number of times the population is evolved, influencing both the precision of the solution and the thoroughness of the search. A higher number of generations allows for a deeper search, potentially improving the quality of the solution. However, this comes at the cost of increased runtime,

requiring a balance between precision and computational efficiency. We choose 20 as the default value for our experiments.

- `population_size`: It specifies the number of solutions in each generation, affecting the diversity and coverage of the search. A larger population increases diversity, enhancing the chances of finding better solutions, but it also demands more computational resources. Therefore, a balance must be struck between diversity and computational efficiency. We choose 35 as the default value for our experiments.

- `num_offsprings`: This parameter determines how many offspring are generated in each iteration. A higher number of offsprings leads to a more extensive exploration of the solution space, potentially improving the quality of the final solution. However, it also increases computational load, so a balance must be found between exploration and computational efficiency. The default value for this parameter is `None`, meaning the number of offsprings is automatically set based on the population size.

- `set_seed`: This parameter controls the random seed used in the algorithm, ensuring reproducibility. By setting the same seed, the algorithm selects the same initial values for the simulation variables in the first generation within the search space.

Next, we pass the configured parameters into the corresponding optimizer within the OpenSBT framework for the specific algorithm being used. By configuring the optimizer with appropriate parameters, such as population size, number of generations, and crossover probability, we are able to fine-tune the algorithm's performance to achieve optimal results.

# 4 Experimental Results

## 4.1 Single-Objective Optimization

One important metric for ensuring the safe flight of quadrotor swarms is avoiding collisions with obstacles during flight. We use `Number of Collisions with Obstacles` as our only fitness function to explore single-objective optimization and we need to investigate whether using metaheuristic search can more effectively determine the optimal placement and diameter of six obstacles, compared to random testing, in order to maximize the total number of collisions between the drones and obstacles, given a fixed target and the initial positions of eight drones.

Experiments with metaheuristic search and random testing were both conducted on a computer equipped with an MX4050 GPU and an Intel Core i7-9850H processor (6 cores, 12 threads), running for approximately 5 hours.

In order to facilitate a more precise analysis of the experimental outcomes, all fitness values in this experiment were squared and then halved, while maintaining their negative sign where applicable. Squaring the values amplifies differences between individual data points, allowing for greater clarity in identifying patterns or anomalies. Halving the result balances this amplification. **Unless otherwise specified, the fitness values specifically refer to the processed values, and the experiments were conducted using Controller1.**

### 4.1.1 Experimental Parameters

- **Simulation Variables:**
  - `quads_obst_size` in the range $0.75 \leq$ `quads_obst_size` $\leq 1.5$.
  - `quads_obstn_position` for $n = 1$ to 6, with $1.6 \leq$ `quads_obstn_position` $\leq 4.799$.
- **Fitness Function:** `Number of Collisions with Obstacles`
- **Generation Number:** 20
- **Population Size:** 35
- **Search Algorithm:** NSGA-II

### 4.1.2 Fitness Values of Different Generations

We can observe from *Table 4.1* that as the generation number increases, the later generations demonstrate a greater ability to identify test cases that result in significantly lower fitness values. This indicates that the algorithm becomes more effective at discovering test cases with a higher number of collisions. In other words, with each successive generation, the algorithm improves in finding configurations that lead to an increased frequency of collisions between the drones and obstacles. However, it is important to note that the performance of the eighth generation actually regressed compared to the seventh, suggesting that while the algorithm generally improves at finding test cases with lower fitness values as the number of generations increases, this trend may not always hold locally.

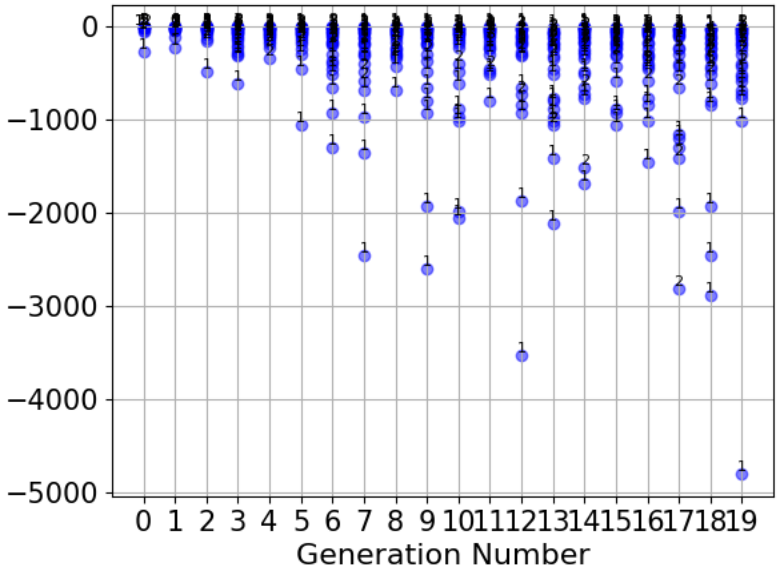
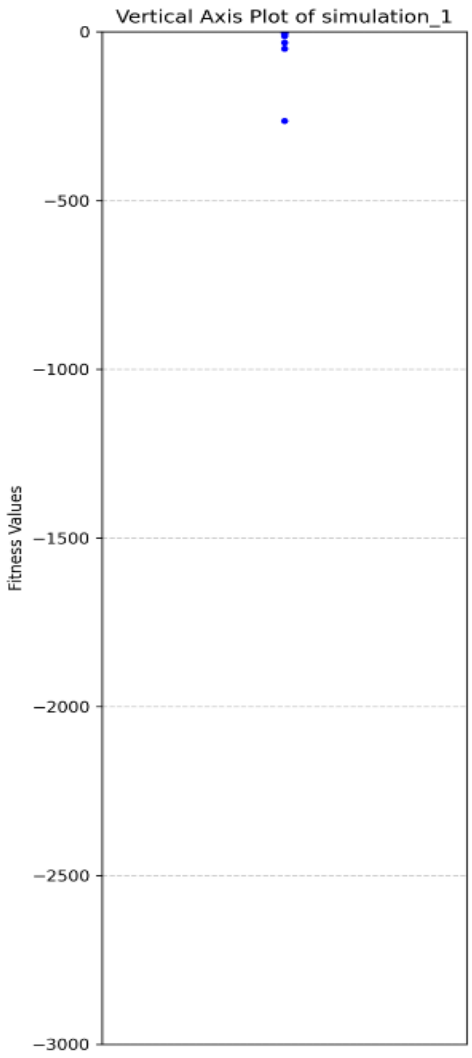



Table 4.1: Fitness Values of Different Generations

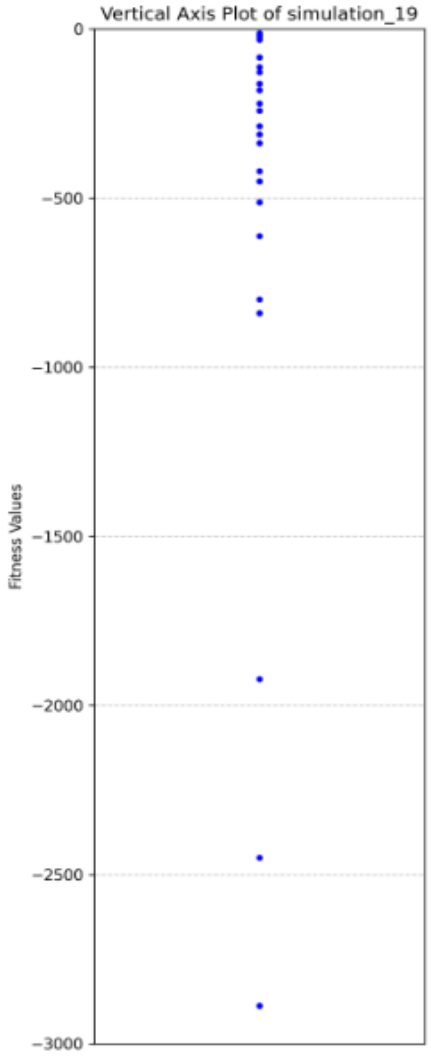
### 4.1.3 Fitness Values Comparison between Generations

The *Table 4.2* below further illustrate the key differences in the distribution of fitness values between the first generation and the 19th generation during the experiment (Because an extreme value appeared in the 20th generation, for clarity, we used the 19th generation instead). In the first generation, most data points are clustered near zero, with only a few points extending to lower fitness values, indicating relatively consistent performance among individuals. In contrast, the vertical axis scatter plot for the 19th generation exhibits a much wider range of fitness values, with some points as low

as near -3000. This suggests greater variability in individual performance in the 19th generation, with some individuals performing significantly worse compared to those in the first generation. Next, we will employ various methods to further process and analyze the combined data from all generations and compare with the results using random testing.




(a) `Fitness Values Distribution of the 1st Generation`

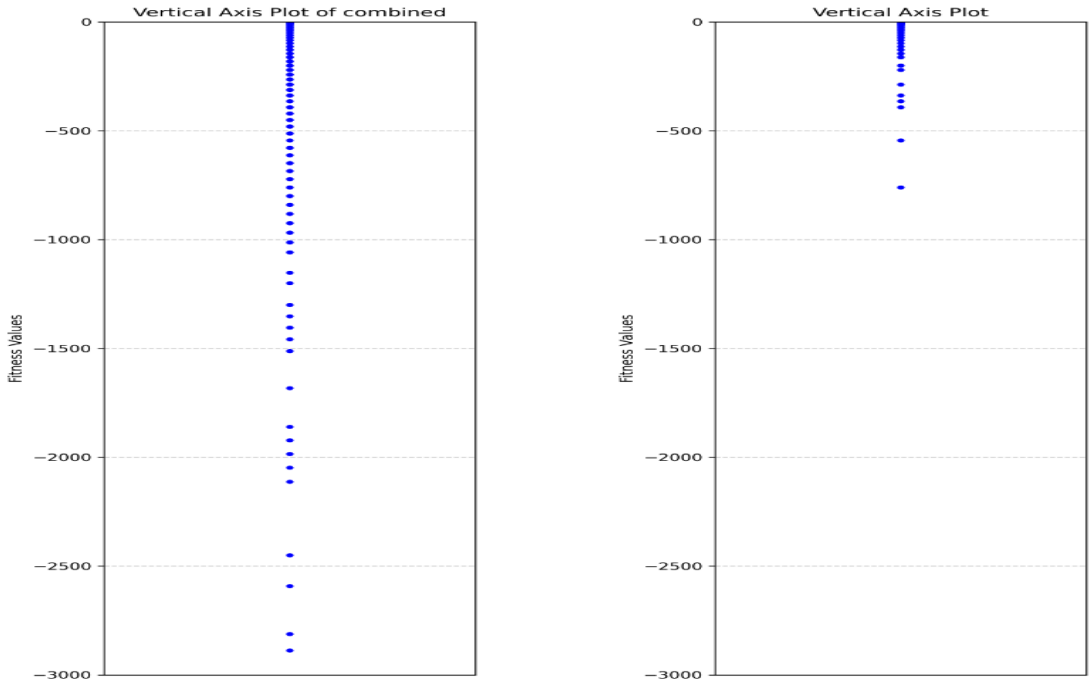



(b) `Fitness Values Distribution of the 19th Generation`

Table 4.2: Fitness Values Comparison between Generations

### 4.1.4 Fitness Values Comparison with Random Testing

This section is closely related to our definition of *effective*, which is central to the objective of our experiments. Readers can refer to the previous sections for the definition of *effective*. *Table 4.3* below presents a comparative analysis of the fitness values both for 700 testing episodes generated with metaheuristic search and random testing, respectively. The fitness values of test cases generated by the random algorithm range from 0 to -800,while most range from 0 to -500. However, with metaheuritic search, we have successfully identified a substantial number of test cases with very low fitness under -1000. The results clearly demonstrate that, the metaheuristic search algorithm consistently identifies a greater number of test cases with lower fitness values. This indicates that the metaheuristic search is significantly more effective in optimizing test case selection.



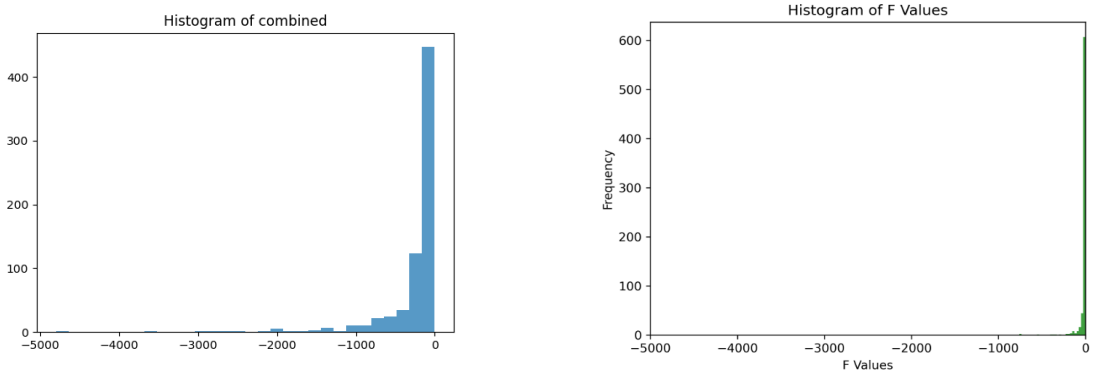

(a) `Fitness Values with Metaeuristic Search`

(b) `Fitness Values with Random Testing`

Table 4.3: Fitness Values Comparison with Random Testing

Now, We perform the comparison using the histogram. A histogram is a graphical representation of the distribution of numerical data, where the data is divided into bins, and the frequency of data points in each bin is represented by the height of the

bar. The advantage of a histogram is that it gives a quick and clear visual indication of the spread and distribution of data across different intervals, making it easy to spot trends, clusters, or outliers. In terms of the histograms from *Table 4.4*, the fitness values with random testing are concentrated around zero, with most values ranging between -200 and 0. In contrast, the histogram with metaheuristic seach shows a much broader distribution of fitness values.



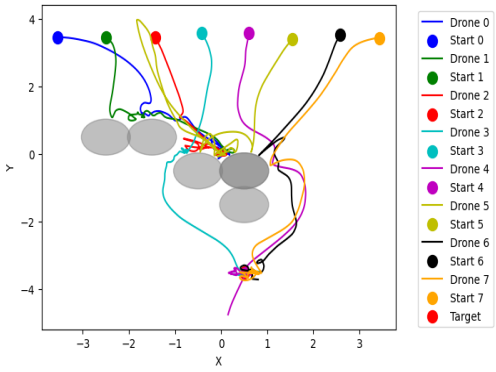

(a) `Fitness Values Distribution with Metaheuristic Search`

(b) `Fitness Values Distribution with Random Testing`

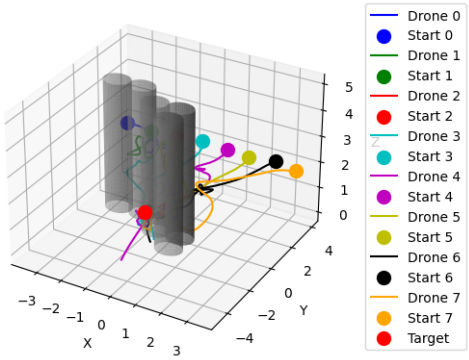
Table 4.4: Comparative Data Representation Using Histograms

### 4.1.5 Critical Test Cases

*Tables 4.5* and *4.6* respectively show a test case with the lowest fitness obtained from metaheuristic search and random testing with collision numbers of 98 and 39. For each table, (a) and (b) represent two-dimensional and three-dimensional visualizations of the simulation episode, respectively, depicting the paths of eight drones navigating from their initial positions through obstacles to approach the goal. We found that they all devised a clever way to place obstacles, ensuring that the drones collided with the obstacles as frequently as possible. It is important to note that since the time limit for each episode is 15 seconds, excessive collisions may result in some drones failing to reach the goal within the time limit, as shown in *Table 4.6*.

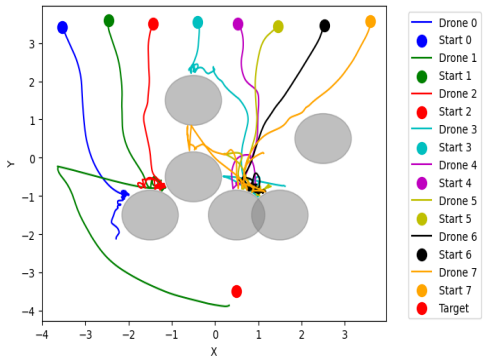



(a) `2D Path Visualization`

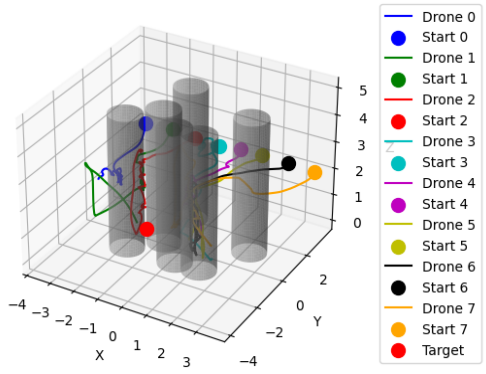



(b) `3D Path Visualization`

Table 4.5: Drone Navigation Paths through Obstacles (Random Testing)



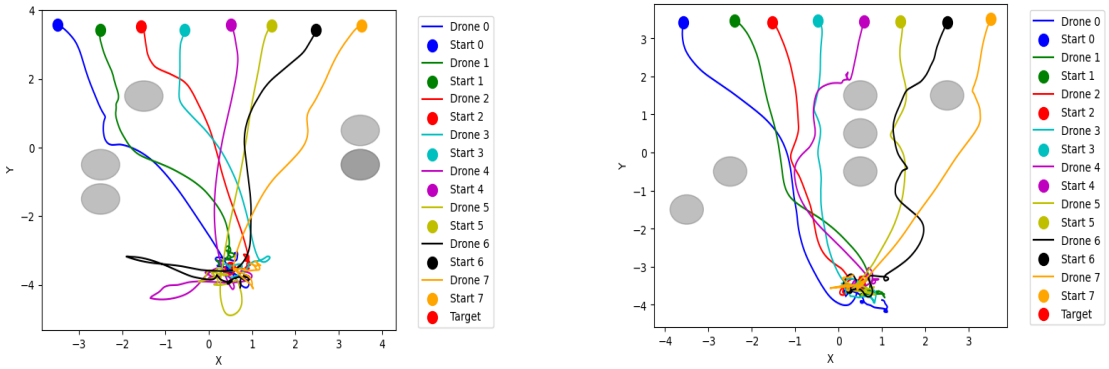

(a) `2D Path Visualization`




(b) `3D Path Visualization`

Table 4.6: Drone Navigation Paths through Obstacles (NSGA-II)

To compare with the placement of obstacles in "a clever way", *Table 4.7* below presents cases where no collisions occurred during the flight while completing the task. It also demonstrates that the controller is able to successfully avoid obstacles in same cases.




(a) `No-Collision Test Case with Random Testing`

(b) `No-Collision Test Case with Metaheuristic Search`

Table 4.7: Drone Navigation Paths without Collisions

### 4.1.6 Performance of Controller2

As mentioned in the previous chapter, **Controller2** possesses only a limited capability for obstacle avoidance. To assess its performance, we applied the same experimental parameters and data analysis methods used for **Controller1**. *Table 4.8* illustrates the fitness values of different generations with **Controller2**, showing how the fitness values change over multiple generations. *Table 4.9* provides a comparison of the fitness values obtained with metaheuristic search and random testing, respectively, by using **Controller2**. The results demonstrated a marked decrease in fitness values when using **Controller2**, which can be attributed to a significant increase in the number of collisions with obstacles. However, despite the observed reduction in fitness, the overall trend in the experimental data remains aligned with the conclusions drawn from **Controller1**. Lastly, *Table 4.10* presents two test cases: (a) illustrates the limited obstacle avoidance capability of **Controller2**, as it fails to slow down and turn when approaching obstacles and instead collides directly with them, and (b) demonstrates that **Controller2** still possesses a basic ability to complete the assigned task to appraching the goal.

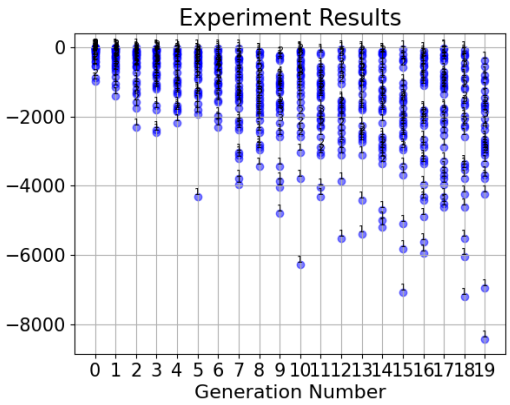
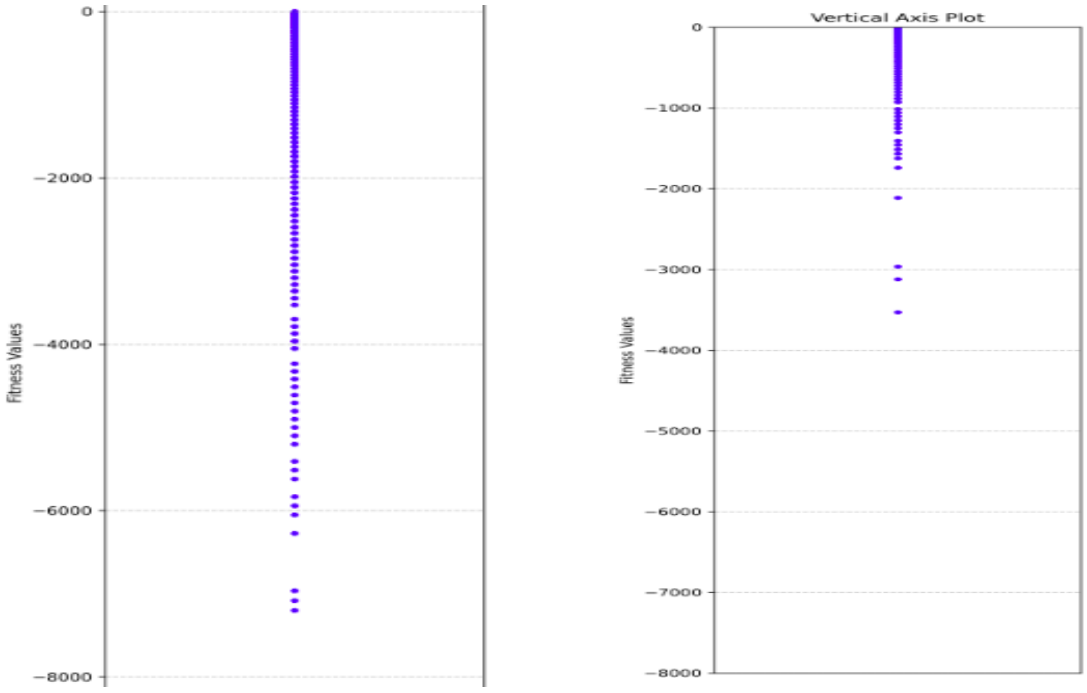


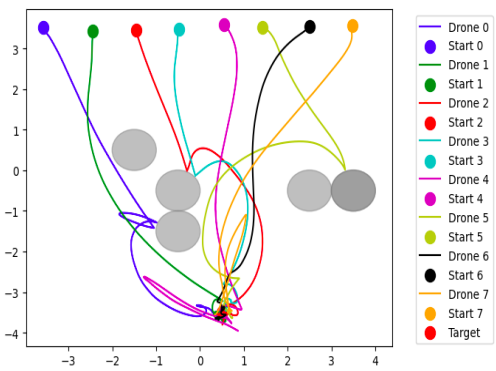

Table 4.8: Fitness Values of Different Generations with Controller2



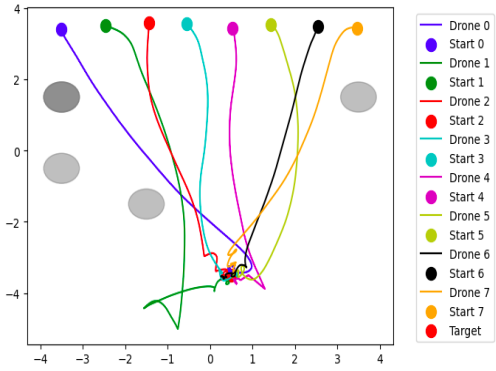

(a) `Fitness Values with Metaheuristic Search`

(b) `Fitness Values with Random Testing`

Table 4.9: Fitness Values Comparison with Random Testing using Controller2




(a) `Limited Obstacle Avoidance Capability of Controller2`




(b) `No-Collision Test Case using Controller2`

Table 4.10: Drone Navigation Paths using Controller2

### 4.1.7 Modifying the Search Configuration

In this section, we do not apply the squaring and division by 2 to the fitness function so the absolute value is exactly the number of collisions. We analyze the experimental results obtained by using different SEED values and offspring numbers. The results demonstrate that the search algorithm remains effective under these configurations. *Table 4.11* illustrates the result of changing the default SEED from 1 to 2. *Table 4.12* shows the results for an offspring number of 20. Additionally, a comparison between the first and last generations is provided in *Table 4.13*, where the optimization is clearly visible. *Table 4.14* presents the results with an offspring number of 10. By comparing *Table 4.12* and *Table 4.13*, it is evident that while a smaller offspring number speeds up the execution of the experiment, it significantly slows down the optimization process, only reaching a optimization of fitness around -40 while it reaches -80 with an offspring of 20.

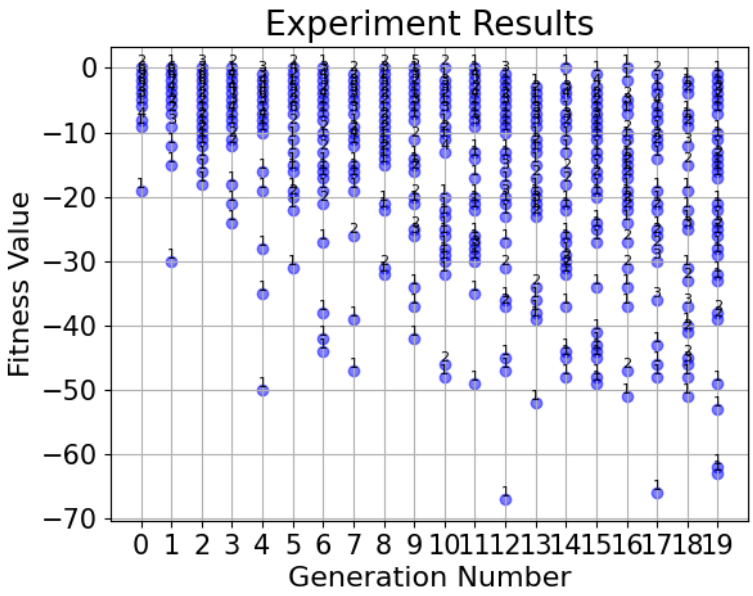



Table 4.11: Fitness Values of Different Generations with SEED = 2

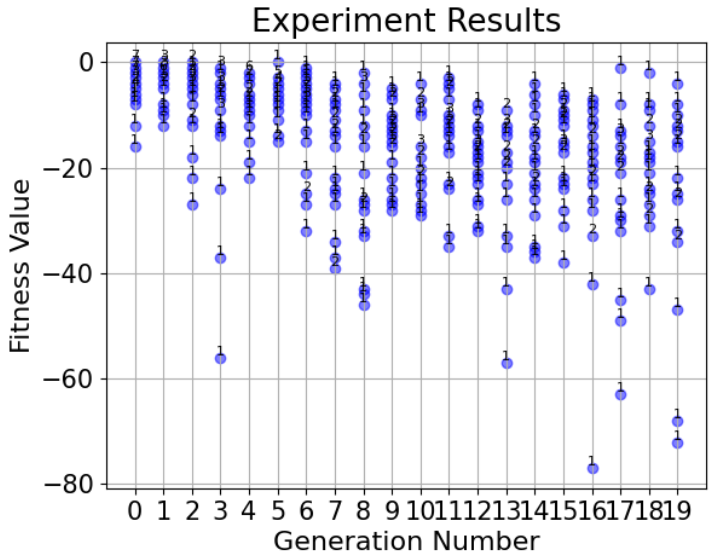
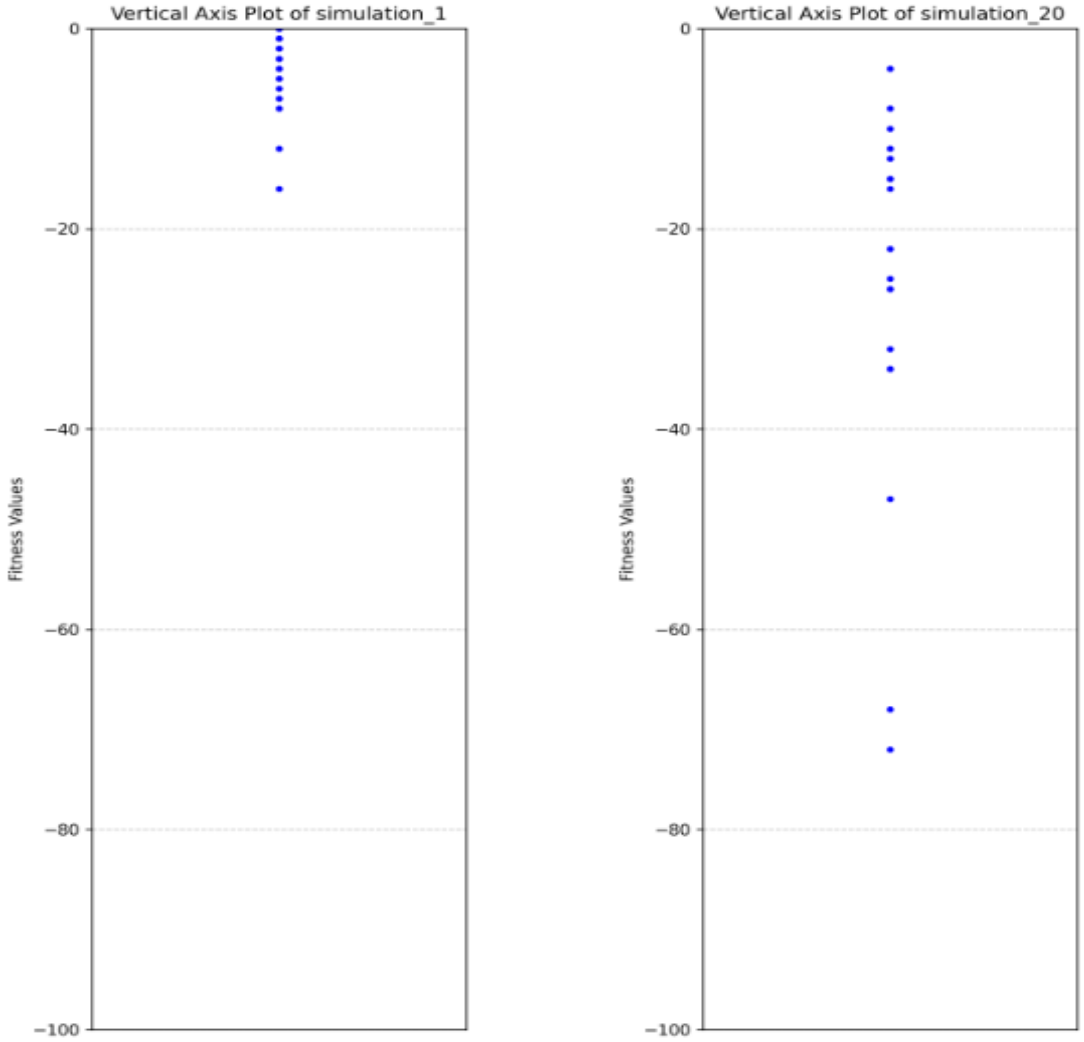



Table 4.12: Fitness Values of Different Generations with Offspring = 20




(a) `Fitness Values Distribution of the 1st Generation with Offspring = 20`

(b) `Fitness Values Distribution of the 20th Generation with Offspring = 20`

Table 4.13: Fitness Values Comparison between Generations with Offspring = 20

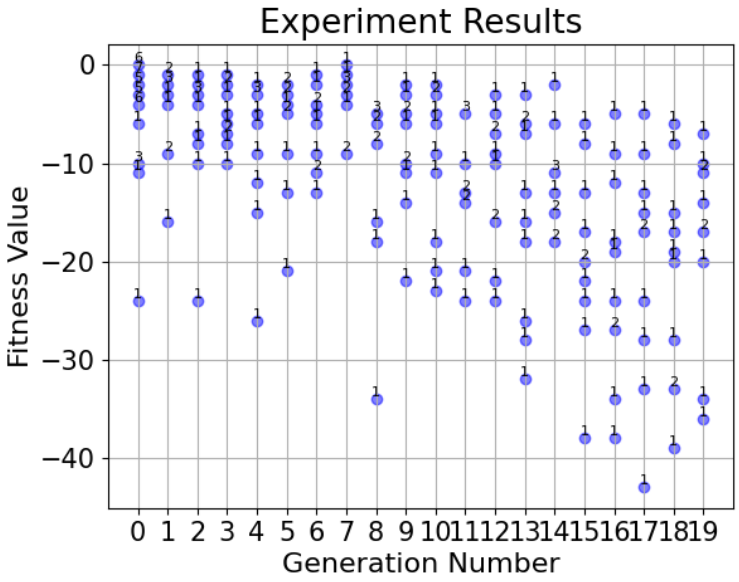



Table 4.14: Fitness Values of Different Generations with Offspring = 10

## 4.2 Multi-Objective Optimization

### 4.2.1 Experimental Parameters

In this experiment, (0, 0) represents the square (-3.5, -3.5) in *Table 3.2* and the coordinate system is established using (0, 0) as the origin.

- **Simulation Variables:**
  - `quads_obst_size` in the range $0.75 \leq$ `quads_obst_size` $\leq 1.5$.
  - `quads_obst1_x_position, quads_obst2_x_position` in the range $1. \leq$ `quads_obst_size` $\leq 8.99$. Take the integer part afterwards, this represents the x-coordinate of two obstacles.
  - `quads_obst1_y_position, quads_obst2_y_position` in the range $3. \leq$ `quads_obst_size` $\leq 6.99$. Take the integer part afterwards, this represents the y-coordinate of two obstacles.
- **Fitness Function:** `Average Distance to Obstacles (of Drone0 at the far top-left initially), Average Distance to Obstacles (of Drone7 at the far top-right initially)`
- **Generation Number:** 20

- **Population Size:** 35
- **Search Algorithm:** NSGA-II

We utilize NSGA-II in multi-objective optimization problems, as it significantly outperforms other multi-objective evolutionary algorithms (MOEAs) in terms of solution diversity and convergence to the Pareto-optimal front. Particularly in real-coded implementations, NSGA-II excels in providing a broader spread of solutions and faster convergence. The algorithm also demonstrates high efficiency in handling constrained multi-objective problems, making it a versatile and powerful tool for solving complex optimization challenges [23].

### 4.2.2 Multi-Objective Optimization

The *Table 4.15* illustrates a comparison between the initial (the 1st generation) and optimized solutions (the 20th generation) for minimizing two objectives: average distance to the obstacles by Drone0 and Drone7, respectively. The Pareto front in the *Table 4.15* illustrates the optimization process. Through optimization, the green points have clearly shifted towards the lower-left region, indicating an improvement in the trade-off between the two objectives: the average distances for Drone0 and Drone7. The initial solutions (red points) are more dispersed, with many points distant from the Pareto front, suggesting that these solutions could be further improved. In contrast, the optimized solutions (green points) form a more compact Pareto front, demonstrating that no solution can improve one objective without worsening the other. The green solutions reflect a range of trade-offs between the two objectives, where some solutions emphasize minimizing the average distance for one drone, while others balance the two objectives differently.

A future research direction is the application of Improved Non-dominated Sorting Genetic Algorithm II (INSGA-II), which enhances the diversity and local search capability by utilizing two populations—an interior population for evolution and an external population to store nondominated solutions—along with a local search mechanism, resulting in superior performance compared to NSGA-II across ten benchmark functions [24].

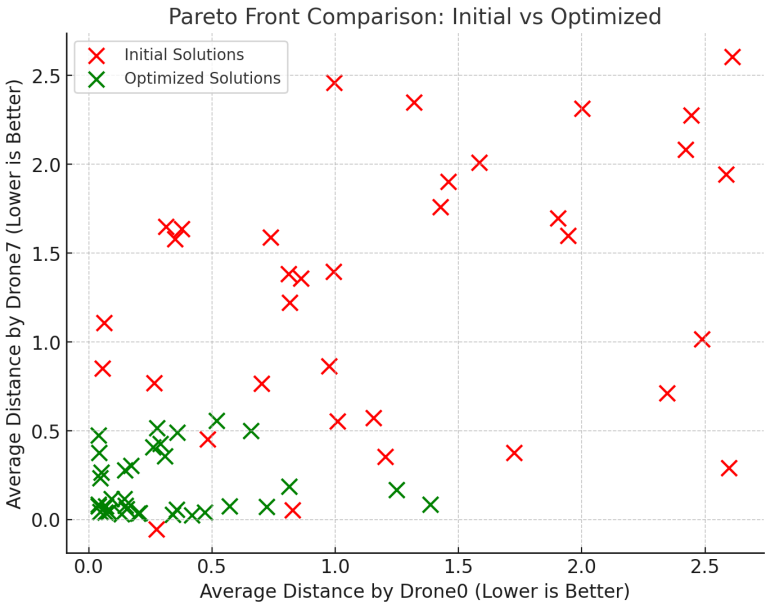



Table 4.15: Multi-objective Optimization Using NSGA-II

# 5 Discussion and Conclusion

After discussing the experimental aspects, this chapter will summarize our contributions to the field, examine the limitations of the proposed methods and experimental sections, and outline potential future work and research directions.

## 5.1 Summary and Contribution

Research on applying DRL to decentralized collaborative drone swarms is still in its early stages. The research we referenced in the experimental section is the first to successfully apply DRL to decentralized collaborative drone swarms [3]. But even though the authors of the research also indicated that they are also conducting in-depth studies to reduce collisions with obstacles [19]. This paper demonstrates that SBT with metaheuristic search can be effective to find critical test cases compare to applying random testing. We first explored the classification, characteristics and applications of UAVs, then discussed the foundations of SBT with metaheuristic search, the functionality of OpenSBT framework, the experimental environment, including the training of various controllers, highlighting their differences and training environments. After that we conducted the experiments, collected and then analyzed the data. We conducted a comprehensive study by exploring both single-objective and multi-objective optimization, defining various fitness functions, and experimenting with different controllers. We applied multiple metaheuristic search algorithms and altered their configurations such as offspring number to thoroughly evaluate performance across different experimental setups.

## 5.2 Limitations and Future Work

### 5.2.1 Limitations of Experiments

In the experimental section, the configuration used for GA is the default configuration. However, the default setup may not always optimize the algorithm's performance or minimize the computation time. Therefore, it becomes crucial to systematically identify the optimal parameter configuration that can improve performance while simultaneously reducing runtime [25], [26]. Furthermore, we tried to apply PSO to

replace GA and the optimization results with PSO were not satisfactory due to the complexity of SUT. Future work will require further exploration of the algorithm and adjustments to the PSO to better fit our DRL model [27].

Another issue is that, despite using the same SEED for mateheuristic seach algorithm, having the controller operate with the same SEED, and applying identical simulation variables values, the behavior of the SUT and the simulation output still vary between runs. Due to the inherent complexity of DRL, we have not yet identified a method to reliably "reproduce" the results. This challenge requires further investigation to address [3]. Additionally, when it comes to the execution of random algorithms, it is important to run multiple trials with different SEEDs to effectively assess the algorithm's performance. The more trials that are conducted, the more reliable the evaluation becomes [28].

Lastly, in our experiments, while the algorithm generally demonstrated significant optimization, there were instances where local regressions occurred. Further research is needed to refine the algorithm to mitigate these regressions and ensure more consistent improvements.

### 5.2.2 Differences between Simulation and the Real World

As previously mentioned, simulation testing plays a critical role in collaborative drone research due to the time-consuming nature of real-world tests and the high costs associated with potential drone damage. However, simulations often rely on simplified physical models, environmental conditions, and interaction rules, which may not fully capture the complexity of real-world scenarios. For instance, communication in real-world situations may experience interference, a factor that is frequently overlooked in simulations.

### 5.2.3 Safety Challenges in Large-Scale Applications

The large-scale deployment of drones presents substantial safety challenges. Although simulations offer detailed testing for particular scenarios, the complexity of real-world environments and extreme conditions—such as sudden weather changes, difficult terrain, and unpredictable human behavior—can surpass the limits of simulations.Therefore, even systems that have undergone extensive testing may find it difficult to guarantee reliable performance in every situation. Moreover, in large-scale applications, even the smallest error rate is intolerable. Achieving absolute safety with deep reinforcement learning-based drones, which operate as black-box systems, is nearly impossible. Therefore, maintaining vigilance and being prepared for potential safety risks is essential.

### 5.2.4 Extending Experiments

Future research could extend the application of search-based testing methods to more complex situations, such as dynamic targets, target switches, and dynamic geometric formations mentioned in [3], and include more simulation variables. Furthermore, while this paper only provides a preliminary exploration of metaheuristic algorithms, future studies could focus on improving these algorithms and tailoring them to address the unique challenges posed by DRL models.

# Abbreviations

**DRL** Deep Reinforcement Learning

**UAVs** Unmanned Aerial Vehicles

**UAV** Unmanned Aerial Vehicle

**SBT** Scenario-Based Testing

**NSGA-II** Non-dominated Sorting Genetic Algorithm II

**GA** Genetic Algorithm

**PSO** Particle Swarm Optimization

**SUT** System Under Test

**PSO-ITC** Particle Swarm Optimization with Increasing Topology Connectivity

**MRS** Multi-Robot Systems

**STARLA** Search-based Testing Approach of Reinforcement Learning Agents

**MOEAs** multi-objective evolutionary algorithms

**INSGA-II** Improved Non-dominated Sorting Genetic Algorithm II

# List of Figures

# List of Tables

# Bibliography


[1] M. Abdelkader, S. Güler, H. Jaleel, and J. S. Shamma, "Aerial swarms: Recent applications and challenges," *Current Robotics Reports*, vol. 2, no. 3, pp. 309–320, 2021, Accessed: 2023-04-05, ISSN: 2662-4087. DOI: 10.1007/s43154-021-00063-4. [Online]. Available: https://doi.org/10.1007/s43154-021-00063-4.

[2] L. He, J. Zhao, and F. Hu, "Distributed multi-agent reinforcement learning for directional uav network control," in *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '23, Orlando, FL, USA: Association for Computing Machinery, 2023, 317–318, ISBN: 9798400701559. DOI: 10.1145/3588195.3595944. [Online]. Available: https://doi.org/10.1145/3588195.3595944.

[3] S. Batra, Z. Huang, A. Petrenko, T. Kumar, A. Molchanov, and G. S. Sukhatme, "Decentralized control of quadrotor swarms with end-to-end deep reinforcement learning," in *5th Conference on Robot Learning, CoRL 2021, 8-11 November 2021, London, England, UK*, ser. Proceedings of Machine Learning Research, 2021. [Online]. Available: https://arxiv.org/abs/2109.07735.

[4] A. Heuillet, F. Couthouis, and N. Díaz-Rodríguez, "Explainability in deep reinforcement learning," *Knowledge-Based Systems*, vol. 214, p. 106 685, 2021, ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2020.106685. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950705120308145.

[5] A. Zolfagharian, M. Abdellatif, L. C. Briand, M. Bagherzadeh, and R. S, "A search-based testing approach for deep reinforcement learning agents," *IEEE Transactions on Software Engineering*, vol. 49, no. 07, pp. 3715–3735, 2023, ISSN: 1939-3520. DOI: 10.1109/TSE.2023.3269804.

[6] A. Farinelli, L. Iocchi, and D. Nardi, "Multirobot systems: A classification focused on coordination," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 34, no. 5, pp. 2015–2028, 2004, ISSN: 1941-0492. DOI: 10.1109/TSMCB.2004.832155.

[7] M. Khelifi and I. Butun, "Swarm unmanned aerial vehicles (suavs): A comprehensive analysis of localization, recent aspects, and future trends," English, *Journal of Sensors*, vol. 2022, 2022. [Online]. Available: http://whitireia.idm.oclc.org/login?url=https://www.proquest.com/scholarly-journals/swarm-unmanned-aerial-vehicles-suavs/docview/2633566739/se-2.

[8] T. Menzel, G. Bagschik, and M. Maurer, "Scenarios for development, test and validation of automated vehicles," *arXiv*, 2018, Accessed: 2022-09-19. [Online]. Available: http://arxiv.org/abs/1801.08598.

[9] G. Zäpfel, R. Braune, and M. Bögl, *Metaheuristic Search Concepts*. Springer, 2010, ISBN: 978-3-642-11343-7. [Online]. Available: https://econpapers.repec.org/RePEc:spr:sprbok:978-3-642-11343-7.

[10] F. Hauer, A. Pretschner, and B. Holzmüller, "Fitness functions for testing automated and autonomous driving systems," in *Computer Safety, Reliability, and Security*, A. Romanovsky, E. Troubitsyna, and F. Bitsch, Eds., ser. Lecture Notes in Computer Science, Cham: Springer International Publishing, 2019, pp. 69–84, ISBN: 978-3-030-26601-1. DOI: 10.1007/978-3-030-26601-1_5.

[11] M. Klischat and M. Althoff, "Generating critical test scenarios for automated vehicles with evolutionary algorithms," in *2019 IEEE Intelligent Vehicles Symposium (IV)*, 2019, pp. 2352–2358. DOI: 10.1109/IVS.2019.8814230. [Online]. Available: https://ieeexplore.ieee.org/document/8814230.

[12] C. Neurohr, L. Westhofen, T. Henning, T. de Graaff, E. Möhlmann, and E. Böde, "Fundamental considerations around scenario-based testing for automated driving," in *2020 IEEE Intelligent Vehicles Symposium (IV)*, 2020, pp. 121–127. DOI: 10.1109/IV47402.2020.9304823. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9304823.

[13] T. Sud, "Scenario-based testing for autonomous vehicles," *TUV Sud Reports*, 2024. [Online]. Available: https://www.tuvsud.com/en-us/industries/mobility-and-automotive/automotive-and-oem/autonomous-driving/assessment-of-automated-vehicles-with-scenario-based-testing.

[14] P. Projekt, *Logical and concrete scenario descriptions for autonomous vehicles*, Accessed: 2024-09-17, 2020. [Online]. Available: https://www.pegasusprojekt.de/files/tmpl/PDF-Symposium/04_Scenario-Description.pdf.

[15] IEEE, "Genetic algorithms," *IEEE Xplore*, pp. 1–10, 1995. [Online]. Available: https://ieeexplore.ieee.org/document/538609.

[16] W. H. Lim and N. Isa, "Particle swarm optimization with increasing topology connectivity," *Eng. Appl. Artif. Intell.*, vol. 27, pp. 80–102, 2014. DOI: 10.1016/j.engappai.2013.09.011.

[17] M. E. H. Pedersen and A. Chipperfield, "Simplifying particle swarm optimization," *Appl. Soft Comput.*, vol. 10, pp. 618–628, 2010. DOI: 10.1016/j.asoc.2009.08.029.

[18] L. Sorokin, T. Munaro, D. Safin, B. Liao, and A. Molin, "Opensbt: A modular framework for search-based testing of automated driving systems," *ArXiv*, vol. abs/2306.10296, 2023. DOI: 10.48550/arXiv.2306.10296.

[19] SampleFactory, *Swarm rl environment integration*, Accessed: 2024-09-20, 2024. [Online]. Available: https://www.samplefactory.dev/09-environment-integrations/swarm-rl/.

[20] P. Rodríguez-Mier, M. Mucientes, and M. Lama, "Automatic web service composition with a heuristic-based search algorithm," Aug. 2011, pp. 81 –88. DOI: 10.1109/ICWS.2011.89.

[21] R. Sarker and M. Kazi, "Population size, search space and quality of solution: An experimental study," in *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, vol. 3, 2003, 2011–2018 Vol.3. DOI: 10.1109/CEC.2003.1299920.

[22] J. Brownlee, *Early stopping to avoid overtraining neural network models*, Accessed: 2024-09-21, 2019. [Online]. Available: https://machinelearningmastery.com/early-stopping-to-avoid-overtraining-neural-network-models/.

[23] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002. DOI: 10.1109/4235.996017.

[24] Y. Fu, M. Huang, H. Wang, and G. Jiang, "An improved nsga-ii to solve multiobjective optimization problem," in *The 26th Chinese Control and Decision Conference (2014 CCDC)*, 2014, pp. 1037–1040. DOI: 10.1109/CCDC.2014.6852317.

[25] S. Sarmady, "An investigation on genetic algorithm parameters," *School of Computer Science, Universiti Sains Malaysia*, vol. 126, 2007.

[26] O. Boyabatli and I. Sabuncuoglu, "Parameter selection in genetic algorithms," *Journal of Systemics, Cybernetics and Informatics*, vol. 4, no. 2, pp. 78–83, 2004. [Online]. Available: https://ink.library.smu.edu.sg/lkcsb_research/841/.

[27] Z. L. Wang, T. Ogawa, and Y. Adachi, "Influence of algorithm parameters of bayesian optimization, genetic algorithm, and particle swarm optimization on their optimization performance," *Advanced Theory and Simulations*, vol. 2, no. 11, p. 1 900 110, 2019. DOI: 10.1002/adts.201900110. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/adts.201900110.

[28] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, 1–10, ISBN: 9781450304450. DOI: 10.1145/1985793.1985795. [Online]. Available: https://doi.org/10.1145/1985793.1985795.