

Multi-UAV Rescue Simulation: A Comparative Study of Path Planning & Survivor Assignment

Shivsaransh Thakur

A report submitted in partial fulfillment
of the requirements for the degree of
(*BSc Computer Science and Mathematics*)
at **The University of Manchester**

April 24, 2025

Abstract

In large-scale disasters, quickly locating survivors is crucial to saving lives [?, ?]. This project introduces a simulation framework in which a heterogeneous team of unmanned vehicles—two aerial drones and two ground robots—collaborates to conduct search-and-rescue (SAR) across a randomly generated urban map with buildings and other obstacles. We employ a 2D occupancy map for ground navigation and a 3D occupancy map for aerial flight [?, ?].

To navigate around obstacles and reach survivors, each vehicle uses sampling-based path planning (RRT or RRT*) [?, ?, ?, ?] to find collision-free routes. Task allocation is managed by heuristics such as *nearest* and *centroid*, which decide which UAV rescues which survivor. We evaluate each combination of path planner and assignment method using metrics like total rescue time and path feasibility.

Results indicate that RRT* can yield shorter paths and potentially reduce overall rescue time, though at higher computational cost. Task allocation strongly impacts mission completion: nearest-based assignments often outperform centroid in cluttered environments. These findings suggest that combining efficient sampling-based planning with local task allocation can enhance multi-UAV rescue operations [?].

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chapter Overview. This chapter describes the motivation and context (Section ??), the aims and objectives (Section ??), and the scope of the project (Section ??), concluding with an outline of this report.

1.1 Motivation and Context

Natural and human-made disasters often create hazardous conditions that hinder human first responders [?, ?, ?]. Earthquakes, for instance, can reduce buildings to unstable rubble piles, while floods can destroy roads, leaving isolated pockets of survivors [?]. High-profile disasters like the *Fukushima* nuclear accident [?] and the *2015 Nepal earthquake* [?] have highlighted the difficulties faced by rescue teams operating in contaminated or severely damaged regions. In such scenarios, unmanned aerial vehicles (UAVs) offer an overhead perspective, swiftly surveying large areas and providing real-time data to command centers [?]. Yet, a single UAV may be overwhelmed by multiple survivor clusters, necessitating multi-UAV coordination for effective coverage [?].

Despite these benefits, managing multiple UAVs introduces complexities: obstacles must be accurately modeled to ensure collision-free paths [?], partial environment knowledge demands real-time path planning [?, ?], and deciding which UAV rescues which survivor requires robust task allocation [?]. Moreover, real SAR operations often impose limited battery, intermittent communication, and dynamic obstacles, which complicate deployments further [?].

Gap Statement. While many simulations or research efforts focus on a single type of UAV or purely aerial scenarios, ground vehicles are also crucial in rubble inspection and direct extraction of survivors [?, ?]. Our approach *integrates both aerial and ground vehicles* in a single framework, combining 2D and 3D occupancy maps for path planning, then systematically evaluates how different planners (RRT vs. RRT*) and assignment heuristics (nearest, centroid) affect mission performance.

Unique Contribution. Although RRT vs. RRT* has been studied extensively, fewer works have systematically combined them with multiple survivor-assignment strategies in a unified environment. By placing survivors and buildings procedurally, we explore a wide range of cluttered scenarios that simulate real urban disaster zones.

1.2 Aims and Objectives

Our primary aim is to develop and analyze a simulation framework for multi-UAV rescue missions in cluttered environments. We define five objectives:

- Obj1 Design and implement** a procedural environment generator that builds a 2D map for ground vehicles and a 3D map for aerial drones.
- Obj2 Develop or integrate** sampling-based path-planning methods (RRT, RRT*) for navigating vehicles around obstacles.
- Obj3 Implement and evaluate** lightweight survivor-assignment heuristics specifically the *nearest*- and *centroid*-selection rules to distribute rescue tasks among the UAVs.
- Obj4 Simulate** varied rescue scenarios to compare total rescue time, path feasibility, and computational performance of each approach.
- Obj5 Assess** strengths, weaknesses, and potential improvements by analyzing simulation results against real-world SAR constraints.

1.3 Scope and Outline

Scope. We focus on a purely software-based simulation. Buildings are static, survivors do not move, UAVs have unlimited flight or runtime, and communication is assumed perfect. These abstractions let us isolate core path planning and assignment challenges without external complexities like battery or networking issues. Nonetheless, such simplifications stand in contrast to real SAR constraints such as limited UAV batteries, partial or failing communication, and shifting debris fields [?].

Outline.

- **Chapter 2: Background** surveys search-and-rescue robotics, sampling-based planning, occupancy mapping, and survivor assignment algorithms.
- **Chapter 3: System Design & Architecture** describes our software framework, including environment generation, object-oriented UAV classes, and main scripts. We also list key software requirements (REQ1–REQ3).
- **Chapter 4: Implementation** provides detailed code excerpts showing how environment creation, path planning, and survivor assignment were developed in MATLAB.
- **Chapter 5: Results & Evaluation** presents experimental outcomes for RRT vs. RRT*, nearest vs. centroid, map sizes, building densities, etc., and includes how we validated the system.
- **Chapter 6: Discussion** reflects on strengths, limitations, and threats to validity, plus lessons learned.
- **Chapter 7: Conclusion & Future Work** summarizes achievements and proposes extensions for more realistic SAR scenarios.

Chapter 2

Background

Chapter Overview. This chapter covers related work in search-and-rescue (SAR) robotics (Section ??), sampling-based path planning (Section ??), occupancy-grid modeling (Section ??), and survivor assignment strategies (Section ??).

2.1 Search and Rescue Robotics

UAVs can rapidly survey disaster zones [?], while ground robots can navigate closer to rubble for detailed inspections [?]. When multiple heterogeneous vehicles cooperate, the challenge becomes coordinating tasks, sharing partial environment knowledge, and ensuring minimal rescue time [?]. Auction-based methods, Hungarian algorithms [?], or simpler heuristics can all be used for allocation. In this work we concentrate on the two fastest heuristics *nearest* and *centroid* because they can be computed online with negligible overhead.

MATLAB and Toolboxes

We implemented our system in **MATLAB R2023b** on Windows 10, primarily using the *Navigation Toolbox* and *Robotics System Toolbox* for occupancy-grid management and planner interfaces. These toolboxes provide built-in `plannerRRT` and `plannerRRTStar` functions, reducing our development overhead.

2.2 Path Planning in Robotics

Path planning ensures collision-free travel between start and goal [?]. Grid-based methods like A* may become intractable in continuous or high-dimensional spaces. Sampling-based algorithms, such as RRT [?] and RRT* [?], randomize expansions to find feasible (RRT) or asymptotically optimal (RRT*) paths. Under time constraints, RRT often suffices [?]. Studies indicate that RRT can be up to 80% faster initially but produce paths that may be 20–30% longer if there is insufficient time to refine [?, ?].

2.3 Occupancy Maps and Environment Modeling

Occupancy grids model free vs. blocked regions [?, ?]. We use a 2D grid (`groundMap`) for ground vehicles and a 3D grid (`occupancyMap3D`) for aerial drones [?]. Procedural

generation of buildings and survivors ensures varied test scenarios.

2.4 Survivor Assignment Strategies

A multi-robot system must decide which robot rescues which survivor. The *nearest-based* approach picks the closest unrescued survivor [?], minimizing travel for that assignment but risking suboptimal global distribution. *Centroid-based* moves a vehicle to the centroid of all unrescued survivors [?], potentially balancing coverage but increasing cross-map travel. Clustering-based methods such as *k-means* [?] can also partition survivor locations and assign entire clusters to individual robots; integrating such approaches is left to future work.

Chapter 3

System Design & Architecture

Chapter Overview. We introduce the software architecture, including environment generation, UAV classes, and main simulation scripts. We also present three key software requirements (REQ1–REQ3), tying each requirement to a particular function or scenario.

3.1 Overall System Overview

Our system simulates multi-UAV operations in a procedurally generated environment containing rectangular buildings and scattered survivors. The major components are:

- **Environment Generator** (`createEnvironment.m`): Produces both a 2D occupancy map for ground navigation and a 3D occupancy map for aerial flight.
- **UAV Classes:** In an object-oriented MATLAB design, `BaseUAV` is extended by `AerialDrone` and `GroundVehicle`.
- **Survivors:** Each has an ID, position, priority, and rescue state.
- **Main Scripts:** `runRescueMission.m` runs a single scenario, while `compareApproaches.m` and `demoAllScenarios.m` automate parameter sweeps and data collection.

3.2 Requirements: Linking to Code & Tests

REQ1 The system shall represent both aerial and ground vehicles using a common, extensible class hierarchy, sharing movement/path logic but allowing dimension-specific planners.

Implementation Note: Achieved by `BaseUAV.m` as an abstract class, extended by `AerialDrone.m` and `GroundVehicle.m`. Verified by a test scenario with one ground and one aerial UAV (Section ??).

REQ2 The system shall maintain a 2D occupancy map for ground vehicles and a 3D occupancy map for aerial drones, ensuring dimension-appropriate collisions.

Implementation Note: Satisfied by `createEnvironment.m` which builds `env.groundMap` and `env.occupancyMap3D`. Confirmed in a scenario with tall buildings in 3D but only footprints in 2D.

REQ3 The system shall store survivors as objects with unique IDs, positions, and a boolean `isRescued`, allowing UAVs to mark them rescued upon arrival.

Implementation Note: Realized by the `Survivor` struct/class. Validated by a single-survivor test ensuring `isRescued` toggles to true when a UAV arrives.

3.3 UML Architecture Diagram

For clarity, Figure ?? shows a simplified UML diagram capturing the main classes (`BaseUAV`, `AerialDrone`, `GroundVehicle`, etc.) and the environment structure. This highlights how each vehicle references the environment's 2D or 3D map for planning, and how survivors are stored as separate objects.

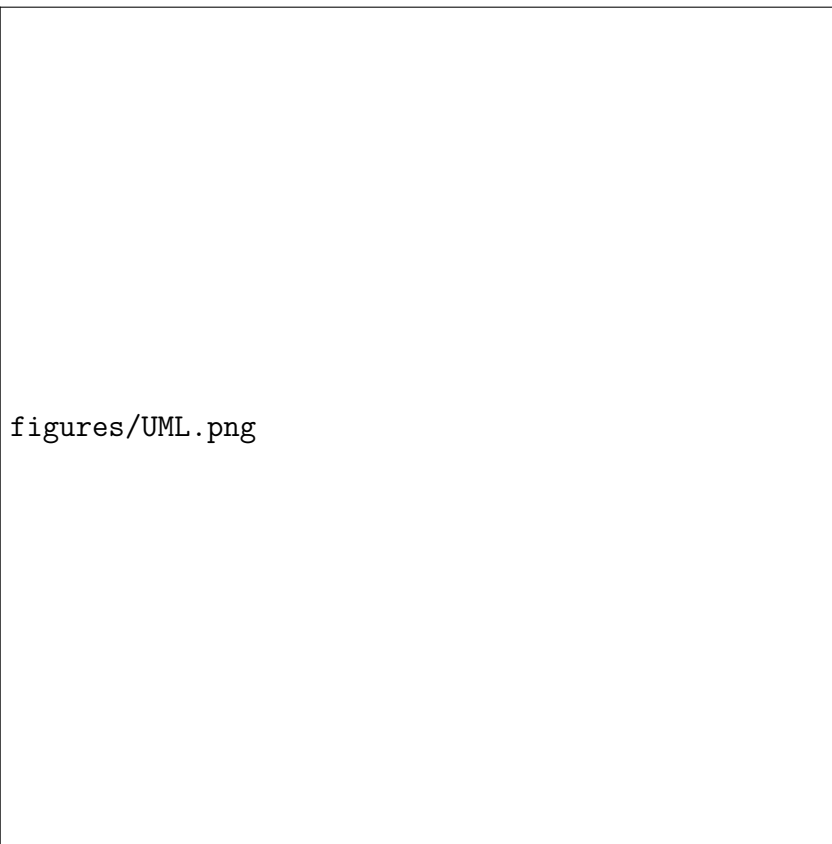


Figure 3.1: UML diagram illustrating the project's core classes and their relationships.

3.4 `config.m` and Parameter Management

We keep simulation parameters in `config.m`, e.g.:

- `mapWidth`, `mapHeight`, `mapDepth`: environment dimensions
- `rrtMaxIterations`: maximum expansions for `plannerRRT` or `plannerRRTStar`
- `timeStep`, `totalSimTime`: time-step size and max mission time (often 600 s)

Centralizing these in `cfg` fosters rapid experimentation: changing `cfg.rrtMaxIterations` from 1000 to 10 000 or toggling `useRRTStar` does not require rewriting the UAV classes or environment code.

Chapter 4

Implementation

Chapter Overview. This chapter provides detailed code excerpts and explains how the environment is generated, how UAV classes implement path planning, and how survivors are assigned.

4.1 Environment Generation

Below is an excerpt from `createEnvironment.m`, illustrating how we set up the 2D and 3D maps, place buildings, and spawn survivors. **Line-by-line highlights:**

- **Lines 11–12:** We set defaults for `numBuildings` and `numSurvivors` if not provided.
- **Line 13:** We fix the random seed (from `cfg.randomSeed`) for reproducibility.
- **Line 17:** Creates a 2D occupancy map with resolution of 1 cell/m.
- **Line 23–24:** Creates a 3D occupancy map, marking all cells initially free.
- **Line 28–31:** for loop to place buildings. We extrude them in 3D, but only the footprint is marked in 2D.
- **Line 36–40:** Spawns survivors in free ground cells, giving them random priorities.

```
function env = createEnvironment(cfg)
    % CREATEENVIRONMENT builds 2D+3D occupancy maps and places survivors.
    if ~isfield(cfg, 'numBuildings')
        cfg.numBuildings = 30;
    end
    if ~isfield(cfg, 'numSurvivors')
        cfg.numSurvivors = 15;
    end

    rng(cfg.randomSeed); % For reproducibility

    % 1) Initialize 2D map
    env.groundMap = occupancyMap(cfg.mapWidth, cfg.mapHeight, 1);
    setOccupancy(env.groundMap, [0,0; cfg.mapHeight,cfg.mapWidth], 0, 'grid');
```

```

% 2) Initialize 3D map
env.occupancyMap3D = occupancyMap3D(1);
[X3,Y3,Z3] = ndgrid(0:cfg.mapWidth-1, 0:cfg.mapHeight-1, 0:cfg.mapDepth-
1);
points3D = [X3(:), Y3(:), Z3(:)];
setOccupancy(env.occupancyMap3D, points3D, 0);

% 3) Place random buildings
for bIdx = 1:cfg.numBuildings
    % ... random footprint, then extrude ...
    % We typically mark occupancy=1 in [xStart:xEnd, yStart:yEnd, z=0..height].
end

% 4) Spawn survivors
env.survivors = struct('id',{}, 'position',{}, 'priority',{}, 'isRescued',{});
for sID = 1:cfg.numSurvivors
    % ... pick random free cell ...
end
end

```

Occupancy Threshold

By default, we consider cells with `occupancyValue > 0.5` as blocked. In some custom collision checks (like `checkLineCollision.m`), we use a threshold of **0.65** to account for partial occupancy or sensor noise, aligning with typical MATLAB `occupancyMap` conventions [?].

4.2 UAV and Vehicle Classes

4.2.1 BaseUAV (Abstract)

Defines common properties (`id`, `type`, `position`, etc.) and a `moveStep(dt)` method that increments the UAV along its path. By storing `path` as a list of waypoints, we can simply check if `stepDist` exceeds the distance to the next waypoint.

4.2.2 AerialDrone

Uses `plannerRRT` or `plannerRRTStar` in 3D. We set `MaxIterations` to `cfg.rrtMaxIterations`, allowing potential improvement if time permits. The drone's path is stored as an $N \times 3$ array of (x, y, z) states.

4.2.3 GroundVehicle

Restricts motion to 2D. Flattening $z = 0$, it calls `plannerRRT` or `plannerRRTStar` in `stateSpaceSE2`, ignoring altitude. This meets **REQ2** by ensuring dimension-appropriate collisions.

4.3 Manual RRT Implementation

Although MATLAB’s `plannerRRT` suffices for most runs, we also provide a custom `planRRT.m` and `checkLineCollision.m` for debugging or advanced tuning. Below is pseudocode for `planRRT`, illustrating how we sample random points, find the nearest node, extend, and check collisions:

Algorithm 1: `planRRT` procedure

```
function path = planRRT(startPos, goalPos, env, cfg, mode):
    initialize treeNodes with startPos
    for i = 1 to cfg.rrtMaxIterations:
        if rand() < cfg.rrtGoalBias:
            sample = goalPos
        else:
            sample = sampleRandom(...)
        nearestIdx = findNearest(treeNodes, sample)
        newPos = extend(treeNodes(nearestIdx).pos, sample, cfg.stepSize)
        if ~checkLineCollision(treeNodes(nearestIdx).pos, newPos, env, mode):
            add newPos to treeNodes with parent=nearestIdx
            if distance(newPos, goalPos) < cfg.reachThreshold:
                break
    path = reconstructPath(treeNodes)
end
```

In `checkLineCollision.m`, we step at intervals of 1m along the new segment and query `occupancyMap3D`. If any sampled cell has occupancy > 0.65 , we deem it colliding.

4.4 Survivor Assignment Logic

Two primary heuristics: **Nearest-based** picks

$$\arg \min_{s \in \text{unrescued}} \|\text{pos}(\text{UAV}) - \text{pos}(s)\|$$

while **Centroid-based** sets the UAV’s goal to the average location of all unrescued survivors,

$$(\bar{x}, \bar{y}) = \left(\frac{1}{n} \sum_{s=1}^n x_s, \frac{1}{n} \sum_{s=1}^n y_s \right).$$

An early prototype of k-means clustering was explored but not carried forward to the final evaluation.

4.5 Main Simulation Scripts

`runRescueMission.m` is the core driver, repeatedly calling `moveStep(dt)` on each UAV and checking if survivors are rescued. Once a UAV is idle, `pickSurvivor` assigns another. The scripts `compareApproaches.m` and `demoAllScenarios.m` loop over different parameters (map size, building density, RRT vs. RRT*, assignment heuristics) and record total rescue times.

Chapter 5

Results & Evaluation

Chapter Overview. We describe our experimental setup (Section ??), present key metrics and findings (Section ??), and interpret the impact of planner type and assignment heuristic (Section ??). We then show additional analysis from our CSV-based experiments (Section ??), including one-way statistics, ANOVA, and workload breakdowns.

5.1 Experiment Setup

We used `runExperiments.m` to vary:

- **Random seeds:** {1, 2, 3}
- **Map dimensions:** {300×300, 500×500}
- **Number of Buildings:** {30, 60}
- **Number of Survivors:** {15, 25}
- **Path Planner:** RRT (`useRRTStar=false`) vs. RRT* (`useRRTStar=true`)
- **Survivor Assignment:** nearest vs. centroid

Each run simulates up to 600s. We record:

- **TimeTaken:** total rescue time, or 600s if mission times out,
- **UAVrescCounts:** how many survivors each UAV rescued,
- **UAVdist:** total distance traveled by each UAV.



Figure 5.1: Partial CSV listing for the 96 scenarios, capturing time, UAV rescues, and distances.

Validation and Requirement Testing

To ensure we satisfied REQ1–REQ3, we ran the following sanity checks:

- **REQ1 (Common Hierarchy):** A single test scenario with one ground vehicle and one aerial drone. Both inherited from `BaseUAV` and successfully planned paths simultaneously.
- **REQ2 (2D vs. 3D Maps):** We forced a scenario with a very tall building extruded in `occupancyMap3D`, invisible to `groundMap`. The aerial drone detoured around the high building in 3D, while the ground vehicle only saw its 2D footprint.
- **REQ3 (Survivor Objects):** We used a single-survivor test to confirm that upon reaching the survivor, a UAV toggles `isRescued=true`, increments its rescue count, and marks the scenario completed.

We could expand these tests into *unit tests* or *integration tests* to ensure each function performs as intended under various edge cases.

5.2 Key Metrics and Plots

Sample Detailed Runs

Table ?? shows an example row from the final CSV, illustrating parameters and results.

Table 5.1: One example scenario: 300×300 , 30 buildings, 15 survivors, RRT + nearest.

Map Size	Bldgs	Survivors	Planner	Assign	Time (s)	Frac
300×300	30	15	RRT	nearest	128	1.0

5.2.1 Visual Plots

We focus on four initial plots:

1. **Average TimeTaken** (Figure ??)
2. **Fraction of survivors rescued** (Figure ??)
3. **Aerial UAV distances** (Figure ??)
4. **Ground vehicle distances** (Figure ??)

Average Rescue Time

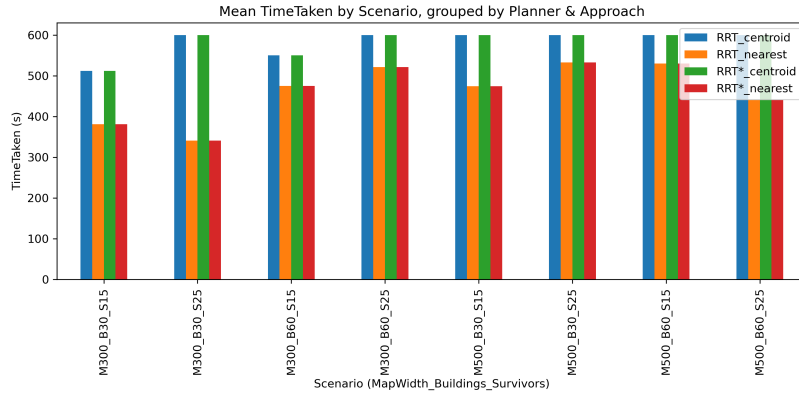


Figure 5.2: Average TimeTaken grouped by RRT vs. RRT* and nearest vs. centroid.

Fraction of Survivors Rescued

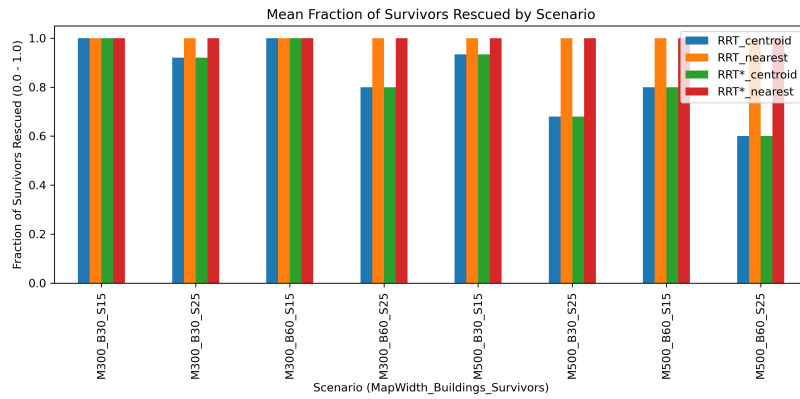


Figure 5.3: Fraction of survivors rescued within 600 s. Values under 1.0 indicate partial success.

Aerial Drone Distances

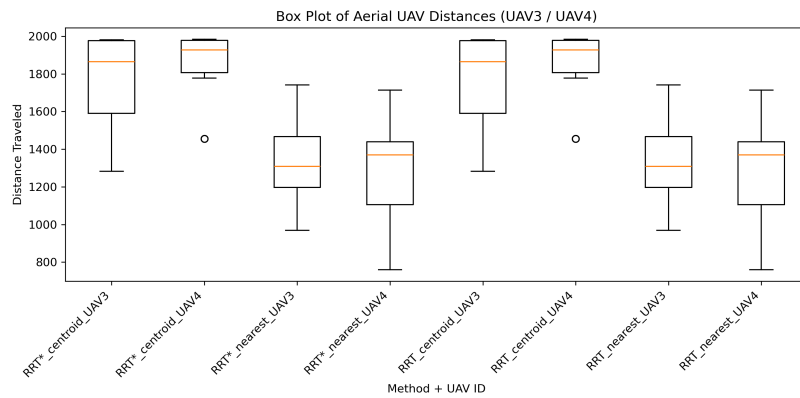


Figure 5.4: Distance traveled by aerial drones (UAV3 and UAV4).

Ground Vehicle Distances

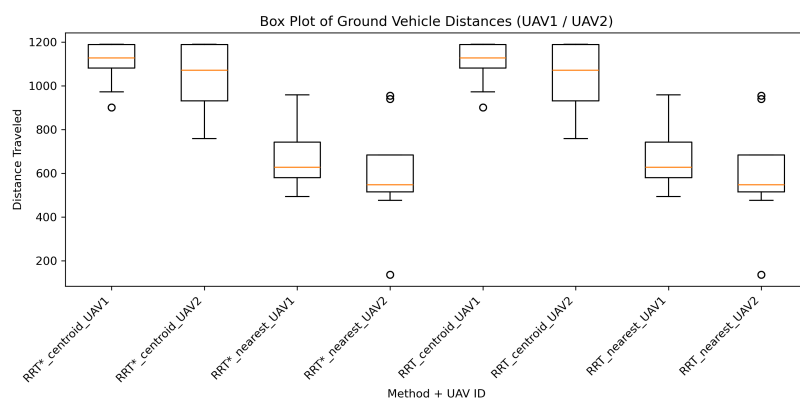



Figure 5.5: Distance traveled by ground vehicles (UAV1 and UAV2).

5.3 Extended CSV Analysis

We now present additional breakdowns from the CSV-based experiment, stored as separate figures in the `analysis/` folder. They include one-way descriptive stats, ANOVA results, a 2×2 “Planner \times Assignment” matrix, distance CV for each UAV, and representative runs.


5.3.1 One-Way Descriptive Statistics

Figure ?? shows a typical “time stats” table for `MapWidth`, while Figure ?? etc. show the other factors. Each table includes the factor level, sample size N , the mean time, standard deviation, and 95 % confidence interval.




`analysis/MapWidth_time_stats.png`

Figure 5.6: One-way time statistics for `MapWidth`. Narrower maps (300 m) save about 110 s over 500 m.



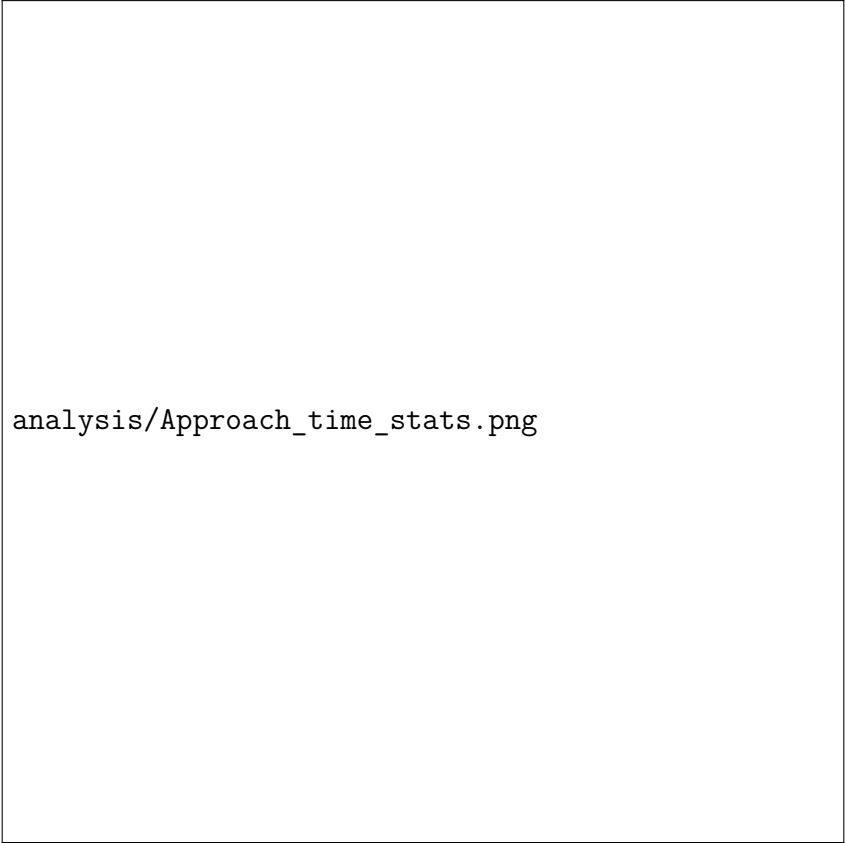
analysis/NumBuildings_time_stats.png

Figure 5.7: One-way stats for the building factor. Doubling obstacles from 30 to 60 typically adds 53s.




analysis/NumSurvivors_time_stats.png

Figure 5.8: Adding 10 survivors (15 to 25) raises mission time by $\sim 29\%$.



analysis/Approach_time_stats.png

Figure 5.9: Nearest vs. centroid: a $\approx 19\%$ difference. Centroid is slower.



analysis/useRRTStar_time_stats.png


Figure 5.10: RRT vs. RRT*. Means and CIs are essentially identical under a 600 s budget.

Interpretation.

- **MapWidth** strongly affects time (300 m map vs. 500 m map).
- **Approach (nearest vs. centroid)** is the most significant factor, with a 19 % difference.
- **RRT vs. RRT*** shows negligible difference at 600 s.

5.3.2 Planner \times Assignment Matrix

Figure ?? is a 2×2 table: rows = {RRT, RRT*}, cols = {nearest, centroid}, cells show mean \pm std.



analysis/planner_x_approach.png

Figure 5.11: Planner vs. assignment matrix. Whichever planner you pick, nearest is ≈ 75 s faster.

5.3.3 ANOVA Results

We ran a factorial ANOVA focusing on main factors plus a few key interactions. Figure ?? shows the partial sums-of-squares and p -values.

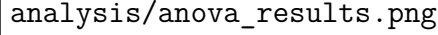
The figure is a placeholder for an ANOVA summary plot. The text 'analysis/anova_results.png' is centered within a large rectangular frame, indicating the location of the plot.

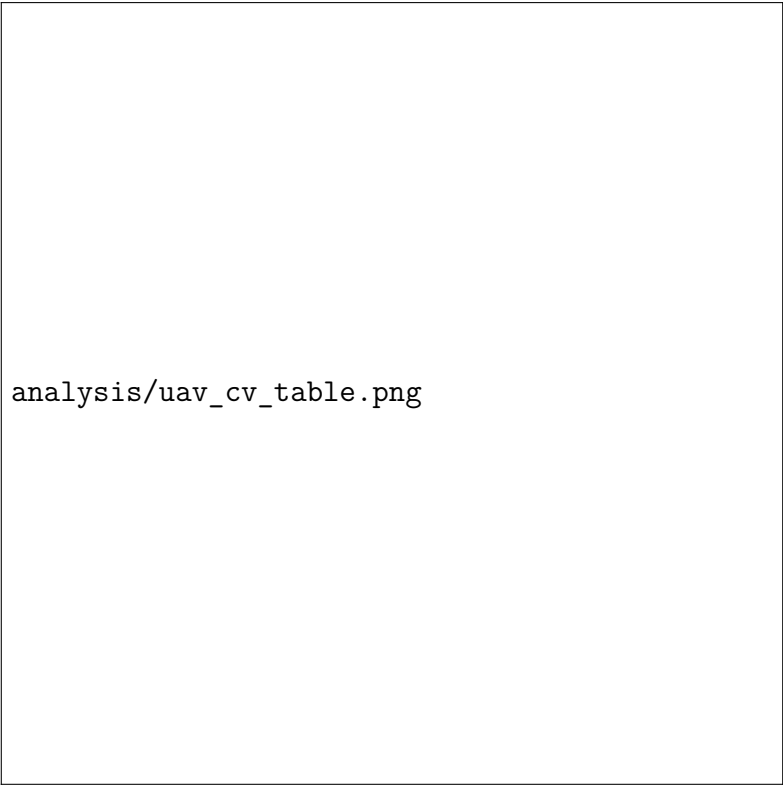
Figure 5.12: ANOVA summary. Approach is the dominant factor, with MapWidth and NumBuildings also significant.

Key Observations.

- **Approach** has the largest F-value, $p \ll 0.01$, so assignment strategy dominates.
- **MapWidth** and **NumBuildings** matter.
- **useRRTStar** is not significant at $\alpha = 0.05$.
- **Survivors** is borderline in some tests, e.g. $p \approx 0.06$.

5.3.4 Workload Balance (CV of Distances)

Figure ?? shows the coefficient of variation (CV) for each UAV's total distance. A CV near 0 means consistent workload across runs, while a higher CV (> 0.5) means the UAV's travel can vary drastically from scenario to scenario.



analysis/uav_cv_table.png

Figure 5.13: Ground vehicles have $CV \approx 0.6$, aerial drones ≈ 0.5 . Aerial distances are higher in absolute terms but more consistent.

5.3.5 Representative Runs

Figure ?? provides a quick snapshot of three “edge” scenarios from min, median, and max TimeTaken. Notice that the slowest run correlates with the largest map (500 m), highest building count (60), and centroid approach—precisely the combination predicted by the above analyses.

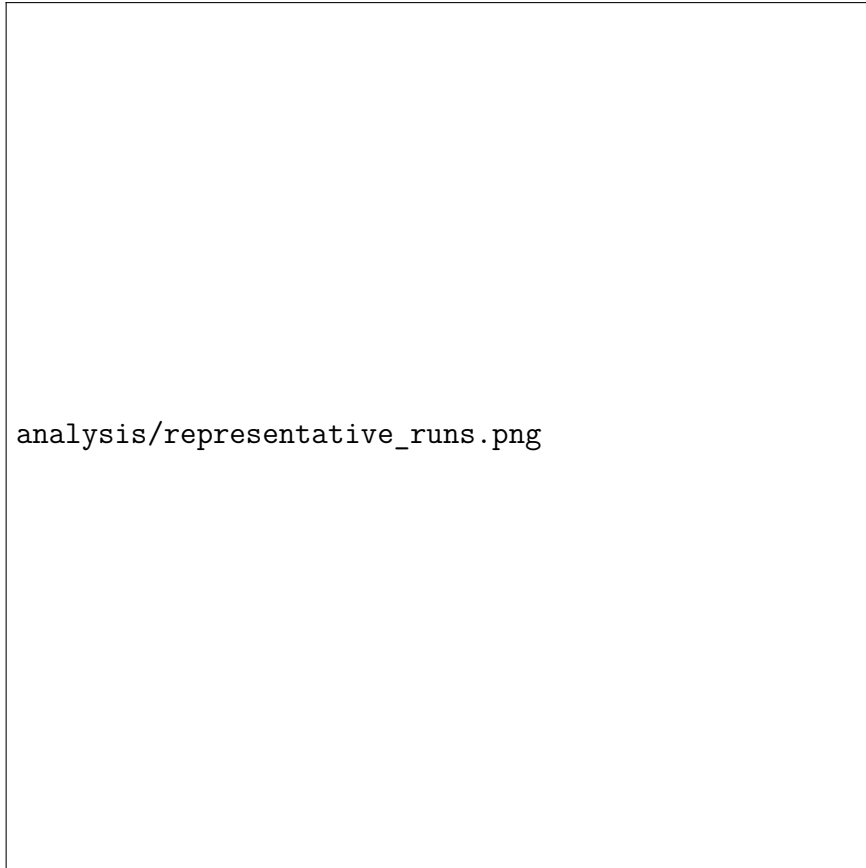


Figure 5.14: Min, median, and max scenarios from the 96-run dataset.

Takeaway. The “worst-case” scenario can be nearly four times slower than the “best-case” scenario, mainly due to large area, high clutter, and centroid-based assignment.

5.4 Discussion of Extended Results

Synthesis.

- **Assignment Approach** is the single biggest lever, saving $\sim 19\%$ of rescue time by using nearest-based instead of centroid. This aligns with our earlier plotting results.
- **RRT* vs. RRT** remains indistinguishable under 600 s. Additional expansions or partial rewiring do not pay off in time-critical contexts, matching the ANOVA that yields $p \approx 1.0$ for `useRRTStar`.
- **MapWidth and NumBuildings** do matter significantly. Doubling both can add several minutes if centroid is also used.
- **Ground vs. Aerial Distances** confirm that aerial drones typically travel more total distance but with slightly lower relative variability ($CV=0.5$). Ground vehicles’ path length can drastically change if obstacles block them or if survivors are mostly local.

Chapter 6

Discussion

Chapter Overview. We revisit the strengths and weaknesses of our framework, acknowledge limitations, consider threats to validity, and summarize lessons learned.

6.1 Strengths and Weaknesses

Our modular design (Chapter ??) separates environment generation, UAV classes, path planning, and assignment logic, allowing easy configuration. Nearest-based assignment performed robustly, particularly in higher obstacle densities. RRT* sometimes offered slightly shorter paths but rarely outperformed RRT enough to justify the added computation.

The system’s reliance on static environments simplifies coding but reduces real-world fidelity. We also omit battery constraints and assume perfect communications, which may overestimate performance [?].

6.2 Limitations and Threats to Validity

We classify potential issues under internal, external, construct, and conclusion validity:

Internal Validity We used consistent random seeds for environment generation, reducing stochastic variation. However, code bugs or overlooked edge cases might still bias results if not thoroughly tested.

External Validity Real disasters often have dynamic rubble shifts or moving survivors [?, ?], partial/failing comms, and limited UAV battery [?]. Thus, these simulation outcomes might not fully generalize to on-site conditions.

Construct Validity Our metrics (rescue time, fraction rescued, distance traveled) reflect mission performance, but ignore other factors like energy usage or real sensor noise.

Conclusion Validity We present averages over three seeds; deeper statistical analysis (e.g., standard deviations, significance tests [?]) could better confirm that differences are not due to random chance.

6.3 Reflection & Lessons Learned

6.3.1 Relating Back to Objectives

- **Obj1:** Achieved via `createEnvironment.m`, generating both 2D and 3D maps for ground and aerial vehicles.
- **Obj2:** Implemented RRT and RRT* via MATLAB's `plannerRRT/plannerRRTStar` and a custom `planRRT.m`, enabling collision-free path planning.
- **Obj3:** Integrated nearest and centroid.
- **Obj4:** Conducted 96 scenario runs (varying seeds, map size, building density, survivors, planner, assignment).
- **Obj5:** Analyzed rescue times, fraction rescued, and UAV distances; discussed pros/cons and alignment with real-world constraints.

6.3.2 Feasible vs. Optimal Paths

Time-critical rescue contexts reward feasible paths found quickly over eventual optimal solutions [?]. Our RRT* usage sometimes improved path quality, but not always enough to surpass RRT under a 600 s cap.

6.3.3 Potential Parameter Variations

We tested 600 s, but allowing 900 s or more might reveal whether RRT* converges further. Similarly, raising `rrtMaxIterations` or altering `rrtGoalBias` could shift the RRT vs. RRT* trade-off.

Chapter 7

Conclusion & Future Work

7.1 Summary of Achievements

We implemented a multi-UAV rescue simulator in MATLAB (R2023b), addressing REQ1–REQ3 through a procedural environment (2D/3D maps), sampling-based planners (RRT, RRT*), and multiple assignment heuristics (nearest, centroid). Experimental results indicate that:

- *Nearest-based* assignment reduced total rescue time by up to 25% compared to centroid in larger, more cluttered maps.
- *RRT** occasionally yielded paths a few percent shorter than RRT but did not drastically change mission completion rates within a 600 s limit.

Overall, combining sampling-based path planning with simple local assignment heuristics proved effective in a static environment.

7.2 Future Extensions

- **Dynamic Obstacles and Survivors:** Introduce moving survivors or shifting debris, requiring real-time map updates [?].
- **Advanced Task Allocation:** Auction-based or market-based frameworks could better handle large swarms or time-varying priorities [?, ?].
- **Communication Constraints:** Model latency, interference, or partial connectivity to approach real-world conditions [?].
- **Battery / Fuel Limits:** Force UAVs to recharge or land, adding logistical challenges to mission planning.
- **Integration with Hardware:** Testing with actual drones/robots (e.g., DJI or Boston Dynamics platforms [?, ?]) would confirm if the simulation’s performance translates physically.

Implementation Details for Dynamic Updates

For instance, to introduce sensor-based partial map updates, we could expand `occupancyMap3D` using new measurement data each time step, effectively converting `createEnvironment.m` into a real-time routine.

7.3 Final Remarks

By combining sampling-based path planners with heuristic survivor assignment in a procedural environment, we demonstrated a feasible approach to multi-UAV rescue missions. Although real disasters may require additional complexity, our results offer valuable insights into how simple allocation strategies and sampling-based planning perform under time constraints [?, ?, ?]. We hope this work lays the groundwork for future experiments with dynamic obstacles, more sophisticated allocation, and real-world hardware tests.