UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Optimizing and Automating Mission Execution in Drone-based Systems

A dissertation submitted in partial fulfillment of the requirements

for the degree of Doctor of Philosophy

Georgios Polychronis

January 2025

UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Optimizing and Automating Mission Execution in Drone-based Systems

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Georgios Polychronis

January 2025

ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΠΟΛΥΤΕΧΝΙΚΗ ΣΧΟΛΗ
ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Βελτιστοποίηση και Αυτοματοποίηση Εκτέλεσης Αποστολών σε Συστήματα με Μη-Επανδρωμένα Εναέρια Οχήματα

Διατριβή η οποία υποβλήθηκε για τη μερική εκπλήρωση
των υποχρεώσεων απόκτησης του Διδακτορικού Διπλώματος

Γεώργιος Πολυχρόνης

Ιανουάριος 2025

UNIVERSITY OF THESSALY
SCHOOL OF ENGINEERING
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

Optimizing and Automating Mission Execution in Drone-based Systems

PhD Dissertation

Georgios Polychronis

Advisory committee

Spyros Lalis, Professor, University of Thessaly (Supervisor)

Christos Antonopoulos, Professor, University of Thessaly

Nikolas Bellas, Professor, University of Thessaly

Examination committee

Spyros Lalis, Professor, University of Thessaly (Supervisor)

Christos Antonopoulos, Professor, University of Thessaly

Nikolas Bellas, Professor, University of Thessaly

Dimitrios Katsaros, Associate Professor, University of Thessaly

Athanasios Korakis, Professor, University of Thessaly

Eleni Karatza, Professor Emeritus, Aristotle University of Thessaloniki

Stathes Hadjiefthymiades, Professor, University of Athens

January 2025

DISCLAIMER ON ACADEMIC ETHICS AND INTELLECTUAL PROPERTY RIGHTS

Being fully aware of the implications of copyright laws, I expressly state that this PH.D. dissertation, as well as the electronic files and source codes developed or modified in the course of this thesis, are solely the product of my personal work and do not infringe any rights of intellectual property, personality and personal data of third parties, do not contain work / contributions of third parties for which the permission of the authors / beneficiaries is required and are not a product of partial or complete plagiarism, while the sources used are limited to the bibliographic references only and meet the rules of scientific citing. The points where I have used ideas, text, files and / or sources of other authors are clearly mentioned in the text with the appropriate citation and the relevant complete reference is included in the bibliographic references section. I also declare that the results of the work have not been used to obtain another degree. I fully, individually and personally undertake all legal and administrative consequences that may arise in the event that it is proven, in the course of time, that this thesis or part of it does not belong to me because it is a product of plagiarism.

The declarant

Georgios Polychronis

Acknowledgements

Completing this PhD was a challenging, yet a very rewarding journey. It would not have been possible without the incredible support, encouragement and guidance of many individuals.

First of all, I would like to thank my supervisor, Spyros Lalis, whose guidance and patience have been irreplaceable throughout this process. His constructive feedback and structured approach have pushed me to continually achieve more. I would also like thank Manos Koutsoubelias, whose insightful advice, and expertise provided great help through this whole journey.

Moreover, I am very grateful to my parents, Apostolos and Katerina, my brother Dimitris and girlfriend Konstantina, for all their encouragement and for believing in my potential. Their support has been invaluable through all these years and without it, I would not have come this far.

Finally I would like to acknowledge that this research has been co-financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH - CREATE - INNOVATE, project PV-Auto-Scout, code T1EDK-02435. It has also been funded in part by the Horizon Europe research and innovation programme of the European Union, under grant agreement no 101092912, project MLSysOps.

PhD Dissertation

Optimizing and Automating Mission Execution in Drone-based Systems

Georgios Polychronis

Abstract

During the last years, unmanned aerial vehicles (drones), polycopters in particular, are used in an increasing number of civilian applications. This trend is expected to continue thanks to rapidly decreasing acquisition costs, simple deployment, high maneuverability and significant customization potential. Namely, drones can be equipped with various sensors and actuators, they can feature Wi-Fi, 4/5G and custom RF modules for the wireless communication with base-stations and servers that are nearby or in the cloud, and they can carry different embedded platforms to perform computations and data processing onboard, according to the requirements of the application at hand.

This thesis focuses on autonomous drones that visit a set of pre-specified locations of interest to perform a sensing task, and, if needed, an additional action (further sensing or some actuation) on the spot. To decide whether such an action is needed, the drones must process the sensed data to detect objects or situations that are relevant for the application at hand. The respective computations can be done locally using an onboard computer, or they can be offloaded to a more powerful server.

We study how to minimize the completion time (makespan) of such missions by (i) reducing the flight time of the drones, (ii) reducing the hovering time over the points of interest due to waiting for the computation to complete, and (iii) offloading computations to nearby edge servers to accelerate data processing. These problems become more challenging when considering the limited autonomy of drones and the need to recharge or switch batteries during the mission. Additionally, the shared servers used by drones to offload computations may have limited computational resources thus requiring suitable scheduling for computation offloading in conjunction with path planning. Furthermore, there may be uncertainty regarding the drones' flight and processing times, introducing changes in the mission plan and server usage. We capture these problems in a formal way, propose solutions and evaluate them through a wide range of simulation experiments, configured using realistic parameters obtained from

corresponding software-in-the-loop setups as well as real drone behavior in field tests.

Last but not least, we propose a modular software architecture for an autonomous remote sensing system based on drones, which fully automates the entire cycle of operation, including handling of user requests, mission planning, take-off, navigation, sensing, landing, charging and processing the data collected by the drones. Notably, the software can split the mission into parts that can be executed in parallel by multiple drones, with adaptive planning in case a drone experiences an issue and must land earlier than anticipated. The complete software has been tested using software-in-the-loop simulations but also in the field using a real drone which takes-off and lands from/on an automatically controlled hangar.

Keywords

Drones; path planning; edge-computing; offload scheduling; mission execution; runtime decisions; system automation; remote sensing; algorithms; system architectures.

Διδακτορική Διατριβή

Βελτιστοποίηση και Αυτοματοποίηση Εκτέλεσης Αποστολών σε Συστήματα με Μη-Επανδρωμένα Εναέρια Οχήματα

Γεώργιος Πολυχρόνης

Περίληψη

Τα τελευταία χρόνια, τα μη επανδρωμένα εναέρια οχήματα (drones), και ιδιαίτερα τα πολυκόπτερα, χρησιμοποιούνται σε όλο και περισσότερες μη στρατιωτικές εφαρμογές. Αυτή η τάση αναμένεται να συνεχιστεί χάρη στη συνεχή πτώση του κόστους κτήσης τους, την ευκολία χρήσης τους, την ικανότητά τους για ελιγμούς και τις σημαντικές δυνατότητες προσαρμογής τους. Συγκεκριμένα, τα drones μπορούν να εξοπλιστούν με διάφορους αισθητήρες και ενεργοποιητές, μπορούν να διαθέτουν Wi-Fi, 4/5G και ειδικές συσκευές RF για την ασύρματη επικοινωνία με σταθμούς βάσης και διακομιστές που βρίσκονται σε κοντινή απόσταση ή στο νέφος (cloud), και μπορούν να διαθέτουν διάφορες ενσωματωμένες πλατφόρμες για την εκτέλεση υπολογισμών και την επεξεργασία δεδομένων, σύμφωνα με τις απαιτήσεις της εκάστοτε εφαρμογής.

Αυτή η διατριβή εστιάζει σε αυτόνομα drones που επισκέπτονται ένα σύνολο προκαθορισμένων τοποθεσιών ενδιαφέροντος για να εκτελέσουν μια εργασία ανίχνευσης και, εάν χρειάζεται, μια πρόσθετη ενέργεια (περαιτέρω ανίχνευση ή κάποια ενεργοποίηση) επιτόπου. Για να αποφασίσουν εάν χρειάζεται μια τέτοια ενέργεια, τα drones πρέπει να επεξεργάζονται τα δεδομένα που συνέλεξαν από τους αισθητήρες για να διακρίνουν αντικείμενα ή καταστάσεις που σχετίζονται με την εφαρμογή. Οι αντίστοιχοι υπολογισμοί μπορούν να γίνουν τοπικά χρησιμοποιώντας έναν ενσωματωμένο υπολογιστή ή μπορούν να εκφορτωθούν σε έναν πιο ισχυρό διακομιστή.

Μελετάμε πώς μπορούμε να ελαχιστοποιήσουμε τον χρόνο ολοκλήρωσης τέτοιων αποστολών (i) μειώνοντας τον χρόνο πτήσης των drones, (ii) μειώνοντας τον χρόνο αιώρησής τους πάνω από τα σημεία ενδιαφέροντος λόγω αναμονής για την ολοκλήρωση των υπολογισμών, και (iii) εκφορτώνοντας υπολογισμούς σε κοντινούς διακομιστές που βρίσκονται στα άκρα του δικτύου (edge) για την επιτάχυνση της επεξεργασίας δεδομένων. Αυτά τα προβλήματα γίνονται πιο δύσκολα όταν λαμβάνεται υπόψην η περιορισμένη αυτονομία των drones

και η ανάγκη επαναφόρτισης ή αλλαγής μπαταριών κατά την διάρκεια της αποστολής. Επιπλέον, οι κοινόχρηστοι διακομιστές που χρησιμοποιούνται από τα drones για την εκφόρτωση υπολογισμών μπορεί να έχουν περιορισμένους υπολογιστικούς πόρους, οπότε απαιτείται κατάλληλος προγραμματισμός για την εκφόρτωση υπολογισμών σε συνδυασμό με τον σχεδιασμό διαδρομών. Ακόμα, ενδέχεται να υπάρχει αβεβαιότητα σχετικά με τους χρόνους πτήσης των drones και της επεξεργασίας δεδομένων, εισάγοντας αλλαγές στο πλάνο της αποστολής και στη χρήση των διακομιστών. Ορίζουμε αυτά τα προβλήματα με επίσημο τρόπο, προτείνουμε λύσεις και τις αξιολογούμε μέσα από ένα ευρύ φάσμα πειραμάτων προσομοίωσης, που έχουν διαμορφωθεί χρησιμοποιώντας ρεαλιστικές παραμέτρους οι οποίες λαμβάνονται από αντίστοιχες προσομοιώσεις όπου χρησιμοποιείται το πραγματικό λογισμικό συστήματος (software-in-the-loop) καθώς και από την πραγματική συμπεριφορά ενός drone σε δοκιμές στο πεδίο.

Τέλος, προτείνουμε μια αρθρωτή αρχιτεκτονική λογισμικού για ένα αυτόνομο σύστημα απομακρυσμένης ανίχνευσης (τηλεπισκόπησης) που χρησιμοποιεί drones, το οποίο αυτοματοποιεί πλήρως ολόκληρο τον κύκλο λειτουργίας, συμπεριλαμβανομένου του χειρισμού των αιτημάτων του χρήστη, του σχεδιασμού της αποστολής, της απογείωσης, της πλοήγησης, της ανίχνευσης, της προσγείωσης, της φόρτισης και τέλος της επεξεργασίας δεδομένων που συλλέγονται από τα drones. Αξίζει να σημειωθεί ότι το λογισμικό μπορεί να χωρίσει την αποστολή σε μέρη που μπορούν να εκτελεστούν παράλληλα από πολλά drones, με δυνατότητα αναπροσαρμογής του πλάνου σε περίπτωση που κάποιο drone αντιμετωπίσει πρόβλημα και πρέπει να προσγειωθεί νωρίτερα από το αναμενόμενο. Το πλήρες λογισμικό έχει δοκιμαστεί χρησιμοποιώντας προσομοιώσεις software-in-the-loop αλλά και στο πεδίο χρησιμοποιώντας ένα πραγματικό drone που απογειώνεται και προσγειώνεται από/σε ένα αυτόματα ελεγχόμενο υπόστεγο (βάση φύλαξης/φόρτισης).

Keywords

Μη επανδρωμένα εναέρια οχήματα, σχεδιασμός μονοπατιών, υπολογισμός στα άκρα του δικτύου, προγραμματισμός εκφόρτωσης υπολογισμών, εκτέλεση αποστολών, λήψη αποφάσεων κατά την εκτέλεση, αυτοματισμός συστημάτων, απομακρυσμένη ανίχνευση / τηλεπισκόπηση, αλγόριθμοι, αρχιτεκτονική συστημάτων.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Context

In recent years, drones have become an increasingly popular trend in various civilian applications. Their contribution to both everyday life and industry has led to their adoption as a key tool in many fields. Photography, agriculture, construction sites, mapping and 3D modeling, search and rescue missions, deliveries are some of these fields where drones have become or are quickly becoming irreplaceable. What is more, the ongoing advancements in battery technology, sensors and telecommunications, are contributing to the growth and increased versatility of drones. These advancements enhance the capabilities of the drones and allow them to perform more complex tasks, further reinforcing their role. The popularity of civilian drones is also reflected in terms of business with significant financial value. According to Business Research [1], in 2023 the global civilian drone market size was valued at 8.89 billion USD and is expected to reach 41.31 billion USD by 2032, growing with an impressive 18.6% compound annual growth rate (CAGR). It is clear that drones are not only becoming a key component of next-generation systems that will support a wide range of civilian applications, but will also play an important role in the economy.

There are different types of drones, with different pros and cons. The most common types are fixed-wing drones and rotary/polycopter drones. Fixed-wing drones resemble typical airplanes, while polycopter drones have multiple vertical propellers (quadcopters and hexacopters being the most popular variants). On the one hand, fixed-wing drones are able to efficiently glide through the air and thus can cover larger distances with less energy, resulting in greater operational autonomy. On the other hand, polycopter drones offer much greater

maneuverability, allowing for better control, which allows them to navigate safely in areas with obstacles or hover steadily above specific points of interest. Furthermore, polycopter drones can take off and land vertically, thus can be easily deployed from various locations without the need for extra infrastructure or a lot of space. In contrast, fixed-wing drones require a runway (or a catapult) to build up sufficient speed to take-off but also to land.

Thanks to software and autopilots provided by various parties, both open-source and commercial, drones can be easily controlled and operated through high-level commands instead of direct manipulation of the drone's flight controls. The typical case is for such control commands to be issued interactively by a person through a dedicated remote control device or an off-the-shelf handheld computer such as a tablet or even a smartphone. However, this still requires direct human involvement, which raises issues of availability, attention-span, casual errors and possibly even unmanageable complexity especially in cases where multiple drones must be engaged concurrently. An alternative approach, which does not require constant manual control, is to issue control commands to the drone using software, through so-called driver programs that communicate with the autopilot through suitable APIs [2]. In the case of autonomous drones, the driver program can run on a companion computer on board the drone itself, which communicates directly with the drone's autopilot over a serial or local network connection. Another option is to run such driver programs on nearby edge servers or ground station, in which case the control commands can be sent to the drone using different wireless technologies such as RF, Wi-Fi or 4G/5G.

Such an automated, program-based control of drones can be highly desirable, both cost-wise and risk-wise, since any human intervention can introduce significant overhead in the deployment and mission execution. Also, manual control can lead to operational mistakes due to tiredness, carelessness or the sheer complexity of the mission at hand. The need for automation is amplified when the drone must execute the same mission periodically, perhaps several times in a day, or when multiple drones need to operate in an area at the same time, where some high-level coordination may be needed to achieve the desired result or to minimize the probability of interference and collisions.

It is important to note that the existence of accurate simulation tools considerably helps in the development of such driver programs or other higher-level software for drone-based systems. This is crucially important given that tests in the field require extensive preparation, are very time-consuming, and also prone to delays or even full cancellations due to unforeseen

and changing weather conditions. Moreover, tests in the real world unavoidably come with the risk of failures and malfunctions, which, in turn, may lead to crashes causing material damages that take time to repair also at a significant cost. Fortunately, thanks to software-in-the-loop (SITL) configurations of autopilots, one may test mission control software for a wide range of scenarios in a cheap, efficient, reproducible and completely safe way, using the same high-level software stack that will run on the real drone.

1.2 Focus

Our work focuses on polycopter drones, primarily due to their flexibility of deployment and maneuverability. Polycopters are also relatively straightforward to design/build so that they can carry a wide range of payloads, such as different sensors (e.g., RGB camera, thermal camera, LIDAR etc) and actuators (e.g. packet/object release mechanisms, sprayers etc). They can also be easily equipped with a variety of computing platforms and communication interfaces, which can be used to run local computations to process data and to interact with other systems.

This renders polycopter drones suitable for a wide variety of missions. Such missions usually consist of visiting certain points of interest where the drone needs to perform one or more tasks. The simplest case is for the drone to collect data via it's on-board sensors, which is then processed offline. However, a mission can be data-driven, in which case the drone may need to perform an additional, follow-up task, depending on data collected. Notably, this decision must be taken at runtime. The required data processing can take place either locally using an onboard computer or be offloaded to a more powerful server at the edge or the cloud.

In the following, we give some indicative examples of such application scenarios from the literature. In [3], the mission is to visit a set of waypoints in order to take pictures and perform a person detection computation. If a person is detected with low confidence then the drone descends for a picture from a lower altitude. If on the other hand, the confidence is high, the image is sent to the ground crew for verification. The crew may subsequently command the drone to go to the location where the person was detected, to take another picture from lower altitude. A sea patrol application scenario is considered in [4] where the drone scans an area to detect people in the sea. If a person is detected, the drone sends a signal to the corresponding

authorities and delivers a supply kit. [5] discusses a precision agriculture application, where a drone is used to identify the type of disease in plants. When the drone detects an infected area, it will go there and use the right type and volume of pesticide to spray. In [6], obscure fires are detected with the usage of a drone, while [7] reports work on designing firefighting drones that can throw fire extinguishing balls. One could combine these capabilities using drones that both detect and control (or even extinguish) fires before any manned vehicle or ground force arrives at the respective locations.

Each of these scenarios includes a sensing and data processing task in order to detect a specific event or situation of interest (a person in need, an infected area in the crops, or a fire). If so, the drone proceeds with an actuation or further sensing task. Note that the action to be taken, if the situation of interest is detected, is typically of high priority and should be ideally performed before continuing with the rest of the mission.

1.3 Challenges

A key objective when planning and executing such missions is to complete the mission at hand as soon as possible. However, in order to achieve this goal one is faced with several challenges.

First of all, polycopters typically have limited autonomy. If the mission is large, the drone will be required to make intermediate stops at specific locations to recharge or swap its batteries. It is important to plan such stops ahead of time, without taking the risk of the drone exhausting its energy reserves mid-flight and having to perform an emergency landing. Aside from causing extra delays in the mission, since someone from the crew must go at the location where the drone has landed to retrieve it, an emergency landing at the wrong location may cause material damages to the drone itself or other infrastructure/property; in the worst case, it might even cause injuries to casual bystanders.

Another aspect that must be addressed are unforeseen delays or increased energy consumption during the mission. Weather conditions such as local wind currents or gusts, maneuvers to avoid obstacles or perhaps other nearby flying drones, as well as battery degradation, are some examples that may cause deviations from the initial plan. In turn, such deviations may render the initial plan suboptimal or even infeasible (thereby leading to an emergency landing, which is highly undesirable for the reasons explained above). In order to address this

issue, the state of the drone must be continuously monitored, and if necessary, the path plan or mission schedule should be dynamically adjusted accordingly.

In data-driven missions, it may not be necessary for the drone to perform a follow-up action at every single point of interest. The drone must have the intelligence to recognize if such an action is indeed needed at a given point of interest, or if it should continue the mission without further delay. Depending on the complexity of the data processing, such computations can be quite time consuming if they take place on the drone's companion board, introducing additional delays in the mission as the drone waits for the result to decide whether a follow-up action is needed. To reduce this delay, the drone can choose to offload the computation to a more powerful nearby server. Still, the server may have limited resources and multiple drones may operate in the same area at the same time, leading to resource pressure and extra delays in the computation at hand. This raises the need for suitable scheduling that ensures fair resource sharing. Moreover, if one considers uncertainties in the drone's flight times (see above), this schedule cannot be static but must be dynamically adapted as needed at runtime.

1.4 Contribution

The overall scope of this thesis concerns the efficient planning and automated execution of the aforementioned types of drone-based applications with the objective of minimizing the mission time while tackling the above challenges. The main contributions of this thesis are as follows:

1. *Path planning for multiple drones.* We consider the limited energy of the drones so that the computed paths are safe and feasible. Additionally, the paths are dynamically adapted to deal with any deviations from the initial plan, due to unforeseen delays or increased energy consumption.

2. *Joint path planning and computation offloading.* We schedule the data processing that must be performed for each drone by choosing whether and where (which edge server to use) to offload the respective computations when visiting each point of interest. The decisions are affected by the location of the drone and the current server load (edge servers have both limited resources and coverage). Path planning and resource usage are interrelated, creating the need to plan schedules that address both problems

jointly. Similar to the pure path-planning problem, when the drones are flying with uncertainties, such schedules must be adapted dynamically.

3. *Stay or go decision.* The above line of work minimizes the mission time by reducing the data processing time via offloading (faster computation). Complementary to this, we tackle the problem of deciding whether to wait for a computation to finish in order to decide if further action is needed at the given point of interest before moving to next point, or start moving to the next point as soon as the sensing is complete. To take good decisions, the drone must effectively learn whether it is likely that a follow-up action will be needed at each point of interest.

4. *Drone-based remote sensing system architecture.* We explore the full automation of drone-based missions for remote sensing systems. More specifically, we design a system that supports the whole cycle of operation, from take-off, mission execution and monitoring, landing, and offline processing, for one or more drones. This is done in a fully transparent way, hiding the complexity of planning and mission execution, without requiring a pilot nor any involvement from the administrator.

1.5 Outline

Chapter 2 investigates the case where multiple drones are used to visit a set of points of interest. We formulate the corresponding path-planning problem in a formal way and propose an offline algorithm that builds paths for each drone of the swarm. Additionally, we propose an online algorithm that dynamically updates these paths in case of unforeseen delays. The goal of this work, is to efficiently build each drone's path and adapt it if needed, so that the mission time is minimized. The first chapter does not consider any computations. This work resulted in the publications [8], [9] and [10].

In Chapter 3 we introduce computations in the mission. More precisely, multiple drones, each performing its own mission, visit a set of points of interest, perform a sensing task and process the sensed data while hovering above the spot. We formally formulate this problem and propose different heuristics that jointly plan the paths and make decisions for the computations of each drone. As discussed, the computations can either be processed locally on the drone, or offloaded to nearby resource-limited edge servers. Similarly to the previous problem, the goal is to minimize the mission time of each drone. Furthermore, we propose an

online decentralized protocol, that adapts the offline schedule produced by these heuristics, in order to deal with any uncertainties during the mission. The work of this chapter, leverage faster computation resources to reduce the time waiting for a computation to complete, and as a result the mission time. This work resulted in the publications [11], [12], [13] and [14]

Chapter 4 focuses in the mission of a single drone. Here, the mission type is the same as in the previous problem. The difference is that, instead of always waiting for the computation to complete in order to decide if a further action is needed, or not, it can continue with the rest of the mission while the data are being processed. For this problem, we propose a learning approach so that the drone can take smart decisions. The drone gains experience from its past missions, learning about the likeliness for an action to be actually needed on each waypoint. As a result it can take informed decisions about whether to wait for a computation to complete or not. Taking good/correct decisions helps the drone to further reduce the waiting time, and so the mission time. This research resulted in publication [15].

Chapter 5 studies the problem of the automation of the mission of one or multiple drones. In this chapter we designed and implemented a system to control the whole operation of one or more drones. The system's architecture is modular allowing its flexible customization according to the application's requirements. In other words, we can easily replace the system's modules so that different algorithms plan the mission or to introduce/implement various behaviors depending the mission's needs. Part of this work is described in publication [16].

Finally, Chapter 6 summarizes the outcomes of the thesis and points to further research directions.

Chapter 2

Drone Path Planning with Flight Time Uncertainties

2.1 Introduction

In this chapter, we focus in missions that require the visitation of some pre-specified points of interest by a fleet of drones (e.g. to gather data by its onboard sensors, deliver goods/supplies or perform another action). The main problem we tackle in this context is to build multiple paths dividing the mission in different parts and to assign each path to one of the drones. The objective is to minimize the total mission time, i.e., to minimize the operation time of the drone that will finish its part of the mission last. Besides the simpler, deterministic path planning problem, we also focus on scenarios where the drones operate with uncertainty on the energy and time that needs to be spent in order to move between two points of interest, requiring the dynamic adaptation of these paths.

The main contributions of this chapter are: (i) We formally formulate and present the problem as a multiple vehicle routing problem (mVRP). The formulation is general and includes both the drones' energy limitations and the mentioned uncertainties. However, to tackle the problem we develop our approach incrementally. (ii) We propose a tournament selection heuristic with a large neighborhood search method that is used to build the drones' initial paths. This algorithm does not consider any energy constraints or uncertainties, so it typically solves a multiple traveling salesman problem (mTSP). For this reason (iii) we evaluate it in a wide variety of benchmarks for the mTSP problem. Then we extend this algorithm to consider the energy and (iv) propose an online heuristic for the dynamic path adaptation

when the drones operate under uncertainties. This algorithm changes the paths and optimizes the plan at runtime considering the actual costs experienced during the mission. (v) Finally we extensively evaluate it over a wide range of scenarios, varying the placement of battery changing stations, the uncertainty of the travel costs and the size of the fleet.

The rest of the chapter is structured as follows. We start by giving a brief overview of related work, in Section 2.2. Then, in Section 2.3, we formulate the problem. In Section 2.4, we describe our proposed offline heuristic, while in Section 2.5 we evaluate its performance. Finally, Section 2.6 presents the proposed online heuristic for the path adaptation and Section 2.7 its evaluation.

2.2 Related Work

2.2.1 Min-max multiple traveling salesman problem

The mTSP has been studied extensively and there is a wide literature on different solutions for it. Indicative surveys can be found in [17],[18], [19]. In this work, we focus on the single depot min-max variant of the problem, where all vehicles (or salesmen since we deal with the problem as a mTSP) start from and have to return back to the same point, and where the objective is to minimize the longest / most costly path. Next, we give an overview of the various algorithms that have been proposed for this problem.

A popular method for tackling the mTSP are genetic algorithms. A genetic algorithm is basically a metaheuristic that is inspired by the principle of natural selection. Genetic algorithms start with an initial population, where each individual represent a different solution to the problem. New solutions can be created either as the result of crossover operations between two different solutions or by performing mutation operations on an individual solution. Given that the population is not allowed to exceed a maximum number of solutions, only the fittest ones are typically selected to remain in the population while the rest are dropped. In [20], the two-part chromosome representation is proposed for the mTSP, where a solution is encoded using a n-length part that is the order of cities together with a m-length part that corresponds to the assignment of cities to the different salesmen. This encoding reduces the search space of the problem compared to other representations. This representation is also used by [21] to devise a new operator that generates a new solution by removing and reinserting genes (cities) at each salesman separately while modifying the second part of the representation in a ran-

dom way. This approach improves the search component of the algorithm. The group-based genetic algorithmic principle was first introduced in [22] in combination with a two-part chromosome, where the first part encodes a solution of the problem and the second part includes the groups of the main part. The mutation and crossover operators are applied in the second part. Based on this approach, [23] proposed a grouping generic algorithm for the mTSP with a suitably adapted solution structure. Finally, [24] propose a grouping genetic algorithm with a different solution representation, where the solutions are represented as m different routes without any ordering or mapping to a specific salesman. This makes it possible to reduce the redundant individuals in a population.

Several researchers have proposed nature-inspired methods. In [25], the authors propose two metaheuristic solutions for the min-max mTSP, an artificial bee colony algorithm and an invasive weed optimization algorithm (IWO). The former is an optimization technique that simulates the foraging behaviour of honey bees. On the other hand, IWO is a technique inspired by the weed colonization and distribution in the ecosystem. The algorithm starts from an initial population of weeds, each representing a solution. Based on the fitness of the currently available weeds, a number of new weeds is produced, which, in turn, join the previous population. However, the number of weeds in the population must remain lower than an upper bound, so there is strong similarity to the genetic algorithms where the fittest individuals stay in the population. [26] and [27] approach the problem using an ant colony optimization algorithm. This is a probabilistic technique, simulating an ant colony and the pheromone used by ants to communicate with each other in order to find good paths toward a food source. Simulated ants move to a customer/city/node randomly, but with a higher chance to pick nodes with high pheromone trails. An approach based on neural networks is proposed in [28]. In [29], the authors hybridize the neural network approach with different metaheuristic techniques such as evolutionary algorithms and ant colony systems.

In [30] a memetic algorithm is proposed, based on variable neighbourhood descend. A memetic algorithm is a hybridization of a genetic algorithm with a local search procedure. Variable neighbourhood descend is a local search where multiple neighbourhoods of a solution are checked until a local minimum is reached. Each neighbourhood corresponds to a different mutation operator. [31] propose a general variable neighbourhood search. The general variable neighbourhood search is a metaheuristic that starts from an initial feasible solution (which at first is also the current solution), improves the current solution with a lo-

cal search procedure (it usually uses multiple operators) and escapes local minimums with a shaking function.

[32] and [33] propose a tabu search, which is a metaheuristic technique to escape local minimums. The solution moves to the best neighbour solution and avoids cycling by keeping a list of forbidden moves, the so called tabu list. [34] propose a task allocation strategy to solve the mTSP. They present an algorithm that first partitions the graph to m subgraphs, and then solve the 1-TSP for each subgraph.

In [35], a comparison is made between the min-sum and min-max mTSP. It is shown that the length of the longest tour in the min-sum problem is at most m times longer than the length of the longest tour in the min-max problem with m vehicles, whereas the total cost is at most m times higher in the min-max than in the min-sum problem. The fact that min-sum mTSP solutions can be highly suboptimal for the min-max mTSP justifies the design of heuristic and metaheuristic algorithms specifically for the latter. But one should also keep in mind that a min-max solution may result to higher aggregated cost compared to a min-sum solution.

The authors in [32], also propose two exact algorithms to tackle the min-max problem. However, given the complexity of the problem, these are not practically applicable when the number of cities is large and there are many alternative paths that can be followed by the salesmen to visit them.

2.2.2 Vehicle routing problem with dynamic aspects

The vehicle routing problem (VRP) has been studied for different dynamic aspects. Indicative surveys can be found in [36, 37, 38]. Most studies focus on the dynamic arrival of new targets, also commonly referred to as customers or customer requests. Some works consider dynamic travel times while others address the problem of known customers with uncertain service needs, which are revealed to the vehicles and the planning system when a customer is actually visited. Below, we provide an overview of indicative approaches that include the aspect of dynamic travel costs/times and are thus closer to the problem we deal in our work.

In [39], the authors address the problem where new customer requests arrive online and there is uncertainty regarding the travel times. Their online approach re-constructs the routes of the vehicles by using the insertion heuristic, every time a new demand arrives or when the travel times change. They further improve the solution with the Or-opt algorithm [40], where

a segment of a route (a number of consecutive customers) is moved to a different position.

A genetic algorithm is proposed for the same problem in [41], where the routes of the vehicles are periodically adjusted taking into account the new information that becomes available regarding changes in the travel times and new customer requests. Another genetic algorithm is presented in [42], which is used to produce the initial schedule as well as to perform any adaptations at a later point in time. In this case, rescheduling occurs when the estimated travel times for an edge is updated (once a vehicle reaches a customer) with the latest information from a dynamic traffic simulator.

In the approach described in [43], every vehicle is allowed to visit the next customer with some tolerance for delays beyond the expected arrival times. If this tolerance is exceeded, the customer is removed from the current route and is reinserted in the best position in the route of another vehicle. In a continuation of this work, the customer can also be added back to the route from where it was removed [44]. In both cases, the paths after the insertions are further improved with the CROSS exchange algorithm, which is proposed in [45] for exchanging entire segments (consecutive customers) between routes. As a further extension, [46] considers a continuous tracking of the vehicle, where unexpected delays are detected earlier, before the tolerance is actually exhausted.

The authors of [47] also deal with the problem of newly arriving requests, in combination with varying travel speeds. In this case, the speed of the vehicle is known as soon as it starts its trip towards the next customer. The routes are adapted using four metaheuristics. The first two approaches are based on a dynamic Variable Neighbourhood Search (VNS) algorithm, based on static and stochastic information, respectively. Two additional metaheuristics are presented, using a Multiple Scenario and a Multiple Plan approach, where multiple solutions are pursued/maintained in parallel, based on static and stochastic costs, respectively. Both approaches use as a search procedure the VNS algorithm.

A wider range of dynamic events is considered in [48], each one possibly necessitating an adaptation of the scheduled routes. More specifically, these events are: late arrival of a vehicle at a customer; timely arrival of the vehicle at a customer that is no longer valid; cancellation of a customer request; generation of a new customer request; break-down of a vehicle; vehicles being stuck in a traffic jam. Two operators are used to optimize the schedule. In the first case, individual customers are removed from a route and it is attempted to insert them in another route. In the second case, customers are exchanged in a pairwise fashion between routes. The

customer insertions are done greedily. The authors also use a secondary objective function to drive the local search so that it focuses on more promising neighbourhoods.

The main difference of our work is that it places a hard constraint on the energy capacity of the vehicles. As a consequence, dynamic changes in travel costs have an impact not only on the optimality but also on the feasibility of the schedule. Also, any adaptations that are made to optimize the routes of the vehicles, must take this constraint into account in order to produce schedules that are actually feasible.

2.2.3 Drone routing and placement problems

The routing and coordination of multiple UAVs have been researched for different application scenarios, objectives and assumptions. Besides routing, some works study the placement of drones in the mission area or on the cooperation between drones and trucks. A recent survey on UAV/drone routing problems can be found in [49]. Next, we briefly outline work that focuses on drones only.

The use of drones to assess the damage due to extreme weather conditions is studied in [50]. Initially, assuming stochastic weather events, the start positions of the drones are decided so as to cover a given target area while minimizing the total setup cost. When an event occurs, the paths to be followed by the drones are planned so as to minimize the operating cost and the mission time. Another placement problem is studied in [51], where a number of provisioning facilities have to be placed in the mission area in order to maximize the delivery of supplies to a set of target locations while respecting the capacity of each facility and the flight range of the drones. During the mission, the drones visit one location at a time, returning each time to a facility, until their battery is exhausted. To cope with energy consumption uncertainty, only a percentage of the drone's battery capacity is utilized. In [52], multiple drones are controlled by different ground stations in order to perform a set of tasks. The objective is to complete the mission under certain constraints, such as fuel tank capacity, flight time, maximum number of drones per ground station, while optimizing various objectives, like the total fuel consumption, the number of drones used, the total travel distance and mission time as well as the risk of the mission. The latter depends on several factors, including having drones that run low on fuel during the mission.

The work in [53] considers drones carrying supplies to several target locations. Apart from the energy constraints, drones also have payload limitations and must return to depot

stations in order to reload. The goal is to minimize the total distance, time or cost of the routes while supplying the target locations giving priority to the ones with the greater needs. A similar problem is studied in [54], where the drone's energy consumption is a function of its current weight (frame, battery and payload) and the objective is to minimize the operational cost given a delivery deadline, or to minimize the delivery time given a hard budget. [55] tackles the same problem for specific delivery time windows while using a more accurate model to estimate the drone's energy consumption. Yet another energy consumption model is introduced in [56], which is derived based on data taken from a power measurement module on a real drone in conjunction with the flight distance, speed and turn angles. The model's estimates are used in a path planning algorithm that minimizes the maximum energy consumption over all drones. The algorithm initially divides the target area in different regions, one for each drone, and then proceeds to plan the path for each drone separately.

There is also work on more dynamic aspects of drone planning/routing. The authors in [57] focus on drones that perform monitoring missions with a required frequency with the objective to minimize travel costs and avoiding penalties due to not meeting the monitoring requirements. Each path is associated with a different monitoring frequency, which may change dynamically in a deterministic or stochastic way. The placement/scheduling of drones in order to monitor both static and moving targets is studied in [58], so as to minimize the number of drones needed or the total energy consumption. The dial-and-ride problem is studied in [59] for drones that can swap their batteries at specific stations, with the objective to avoid having drones that run out of battery and to minimize delivery time and unnecessary movements. A similar problem is considered in [60] for a heterogeneous fleet of drones, taking into account the capacity, the energy and the maximum speed of each drone.

Although some works, like [52] and [51], acknowledge the issue of uncertainty regarding the energy consumption of drones, they do not deal with this issue during the mission.

2.3 Problem Formulation

Our work is motivated by drone-based applications, in particular polycopter drones, such as quadcopters and hexacopters, which have limited autonomy. Still, the problem we study holds for other types of drones or even conventional vehicles as well. Therefore, we formulate the problem as a general multi vehicle routing problem.

2.3.1 Mission topology

The target area where the mission is being executed is modeled as a directed graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. $\mathcal{N}$ is the set of nodes n_i each representing a different point/location, while $\mathcal{E}$ is the set of edges. Each edge $e_{i,j}$ represents the ability of a vehicle to move directly from node n_i to node n_j.

In order for a vehicle to move, it has to spend some of its energy, but it can also gain energy at the so-called depot nodes. These depot nodes represent battery switching or charging/refueling stations. Let $\mathcal{D}$ be the set of depot nodes, $\mathcal{D} \subset \mathcal{N}$. Without loss of generality, we assume that the set of depot nodes $\mathcal{D}$ is disjoint from the set of points of interest, let $\mathcal{V}$, that have to be visited by the vehicles, $\mathcal{D} \cap \mathcal{V} = \emptyset$. The nodes in the graph $\mathcal{G}$ include both the nodes to be visited and the depot nodes, $\mathcal{N} = \mathcal{V} \cup \mathcal{D}$.

2.3.2 Energy costs and gains

Each vehicle has a maximum energy capacity B. This represents the capacity of its battery (or the size of its fuel tank). The vehicle's energy reserves change as it moves between nodes. More specifically, when the vehicle traverses an edge $e_{i,j}$, the remaining energy is reduced. Let $c_{i,j}$ be the energy cost for the hop from n_i to n_j. Conversely, when a vehicle is on a depot node $n_i \in \mathcal{D}$ it restores its energy to the full capacity B. We assume that the depots can always fully restore the energy reserves of any vehicle.

Let b be the energy/battery reserves of the vehicle at a given point in time. Then, the energy that remains available if the vehicle moves from node n_i to n_j is:

$$rem1(b, n_i, n_j) = b - c_{i,j} \tag{2.1}$$

Note that if the start node is a depot, $n_i \in \mathcal{D}$, the vehicle will perform this hop starting with the maximum energy reserves $b = B$. If $rem1(b, n_i, n_j) \leq 0$ then the vehicle has exhausted its energy and stops operating without completing the hop. The energy gain at the destination node n_j (if this is a depot) is not taken into account in Equation 2.1 because it cannot be used to perform the hop in question. However, once the depot has been reached, it can be exploited by the vehicle to restore its energy reserves before attempting the next hop.

2.3.3 Paths and time aspects

We model a path p as a node sequence $p[k], 1 \leq k \leq |p|$, where $p[k]$ is a node of $\mathcal{N}$ and $|p|$ is the number of nodes in p. A path is non-empty if it includes at least one hop, $|p| \geq 2$. Note that in order for $p[k] = n_i \wedge p[k+1] = n_j$ to hold, there has to be a corresponding edge $e_{i,j} \in \mathcal{E}$. In such case we also say that $e_{i,j} \in p$. Furthermore we use the notation $p[k1 : k2]$ to denote the part of p starting from node $p[k1]$ and ending at node $p[k2]$.

Let there be M vehicles that can be used to perform the mission at hand. Then, each vehicle m can follow its assigned paths in parallel/independently to the other vehicles. In the general case, each vehicle m will be assigned several paths, $paths_m[k], 1 \leq k \leq |paths_m|$, where $|paths_m|$ is the number of paths to be followed by vehicle m. Each such path starts from a depot node, $paths_m[k][1] \in \mathcal{D}$, and also ends at a depot, $paths_m[k][|paths_m[k]|] \in \mathcal{D}$. Further, the end node of a path is the same as the start node of the next path, $paths_m[k][|paths_m[k]|] = paths_m[k+1][1]$. The reason for assigning several paths to a vehicle is to make intermediate stops at depot nodes in order to restore its energy reserves so that it can safely visit additional nodes of interest. In the special case where the vehicles do not have energy limitations then, only one path per vehicle is required. Also, the vehicles initially start the mission from a depot and have to return back to a depot after the mission completes. We call the set of all $paths_m, 1 \leq m \leq M$, a plan/schedule s. In other words, a plan (or schedule) s consists of a set of paths along with their assignment to the available vehicles.

Let $t_{i,j}$ be the time needed for a vehicle to traverse $e_{i,j}$. We assume that $t_{i,j}$ is proportional to $c_{i,j}$. Then, the completion time of path p is $d(p) = \sum_{e_{i,j} \in p} t_{i,j}$, and the total time needed for vehicle m to complete all the assigned paths is $d(m) = \sum_{k=1}^{|paths_m|} d(paths_m[k])$. We assume that the restoration of the vehicle's energy reserves at a depot is negligible compared to the travel time and does not affect $d(m)$ in a significant way. Finally, since vehicles travel in parallel to each other, the makespan (or completion time) of the mission is $max_{m=1}^{M} d(m)$. This is the time it takes for the last vehicle to complete all the paths that have been assigned to it based on the plan.

2.3.4 Path feasibility

Based on the above, the remaining energy of the vehicle if it starts with energy reserves b and travels along path p, can be expressed as follows:

$$rem(b,p) = \begin{cases} rem1(b,p[1],p[2]) & |p| = 2 \\ rem(rem1(b,p[1],p[2]),p[2:|p|]) & |p| > 2 \end{cases} \quad (2.2)$$

Namely, if p consists of a single hop, the remaining energy is given by Equation 2.1. If p includes more than one hops, the energy that remains available at the end of p is equal to the remaining energy for the path without the first hop, starting with the energy that remains after taking the first hop of p. As this is the case for a single hop, this does not include the energy gain at the final destination node even if this is a depot.

We say that path p is feasible for initial energy/battery reserves b, if $rem(b, p[1:k]) > 0, \forall k : 1 < k \leq |p|$. In other words, p is feasible if the vehicle will not exhaust its energy in any hop along p. Finally, let $nodes(p)$ denote the set of nodes that are part of p.

2.3.5 Problem

We assume that M vehicles can be used to visit the pre-specified points of interest $\mathcal{V}$. The objective is to build a plan consisting of feasible paths so that all nodes of interest are visited $\cup_{m=1}^{M} nodes(paths_m) = \mathcal{V}$ while minimizing the makespan $max_{m=1}^{M} d(m)$.

Our work also tackles a dynamic version of the problem, where there is some uncertainty about the travel costs that will occur during the mission. More specifically, we assume that the edge costs $c_{i,j}$ follow a known distribution in the range $[c_{i,j}^{min}..c_{i,j}^{max}]$ but the actual cost that will be incurred when a vehicle performs a hop is not known beforehand, when producing the plan. Since $t_{i,j}$ is proportional to $c_{i,j}$, there is also uncertainty regarding the time it takes to perform a hop. An offline solution to this problem can be quite suboptimal. Therefore, we propose an online method that takes planning/adjustment decisions at runtime.

We study the problem for the case where there is a single depot node, let this be n_d. Thus, every path p assigned to a vehicle starts from this node, $p[1] = n_d$, and ends at this node, $p[|p|] = n_d$. Also, in order for the problem to be always solvable, we assume that B is sufficiently large so that, starting from n_d, a vehicle can visit any node $n_i \in \mathcal{V}$, and then return back to n_d, even if every hop over edges $e_{i,j}$ incur the maximum possible cost $c_{i,j}^{max}$.

We tackle the problem in two different ways. In a first approach, we employ an offline algorithm for constructing the paths for each drone. These paths are then followed (without any changes) during mission execution. In a second approach, we start from offline-computed initial paths but these are adapted at runtime based on the true costs that are experienced during

the mission. The two approaches are described in more detail in the following subsections.

2.4 Offline Algorithm

This section describes the proposed offline algorithm. At this point, we assume that drones do not have energy constraints. The heuristic is based on the principle of tournament selection (TS), combined with the large neighborhood search (LNS) method, originally proposed by [61]. Each solution is schedule consisting of M paths, one for each vehicle (since we do not consider any energy limitations). The representation of the routes is similar to the one used in [24], [25] and [30]. As a fitness function for a given solution, we use the inverse of the cost of the most expensive (worst) route. When comparing between two solutions, we prefer the one for which the fitness function returns the larger value.

In the sequel, we present the algorithm in a top-down fashion. We start with the main logic and then proceed to discuss the different functional components of the algorithm in more detail.

2.4.1 Top-level iteration function

The starting point of the algorithm is the TS-LNS() function, described in Algorithm 1. It builds an initial population consisting of $MaxPopSize$ random solutions, and subsequently evolves this population in an iterative fashion.

In each iteration, the fittest solutions from the previous population are kept, decreasing the size of the population by a factor f. For each solution in the remaining population, a large neighborhood search (LNS) is performed.

The iterations are repeated until the size of the population drops to/below a pre-specified threshold $MinPopSize$. The fittest of the remaining solutions is returned as the end result.

2.4.2 Large neighborhood search

The large neighborhood search procedure is described as a separate function LNS() in Algorithm 2. It takes a solution as input and returns as a result another solution, which is produced by trying out a number of so-called mutations. The number of mutations to be performed is a parameter, provided by the top-level TS-LNS() function.

Algorithm 1 TS-LNS algorithm for M drones (option $rmvopt$ sets the node removal method)

function TS-LNS($\mathcal{N}, M, rmvopt$)
  $nullsol \leftarrow \varnothing$
  $pop \leftarrow \{\}$
  repeat M times
    $nullsol = nullsol + \{[n_d, n_d]\}$
  end repeat
  repeat $MaxPopSize$ times
    $pop \leftarrow pop +$ INSERT$(nullsol, \mathcal{N})$
  end repeat
  $sort(pop)$
  $mut \leftarrow$ initLNSMutations()
  repeat
    $pop \leftarrow$ getFittest(pop,size$(pop)/f$)
    for each $sol \in pop$ **do**
      $sol \leftarrow$ LNS$(sol, mut, rmvopt)$
    end for
    $sort(pop)$
    $mut \leftarrow$ adjustLNSMutations(mut)
  until size$(pop) \leq MinPopSize$
  return getFittest($pop, 1$)
end function

Each mutation generates a new solution based on the best solution found up to that point, first by destroying it and then by repairing it. The destruction operation involves the removal of some nodes from their assigned routes, and the repair operation reinserts those nodes to some (other) routes. If the new solution is fitter than the current one, it is adopted as the best solution, which, in turn, will be used as a basis for the remaining mutations.

The node insertion and removal methods used to implement the mutations are discussed separately. Note that LNS() is designed to work using two different node removal methods. The selection is done via the $rvmopt$ parameter, which is set by the user when invoking the top-level TS-LNS() function. In any case, the number of nodes to be removed and then reinserted in every mutation is decided randomly. However, the interval for this random pick is defined as a function of either $|\mathcal{N}|$ or $\sqrt{|\mathcal{N}|}$, depending on the node removal method used.

2.4.3 Node insertion

The node insertion logic is given as a separate function INSERT() in Algorithm 3. This seems to be similar to the approach used in [25].

Briefly, a node is picked randomly from the set of nodes to be incorporated in the solution,

Algorithm 2 LNS method (option $rmvopt$ sets the node removal method)

function LNS($sol, nofmutations, rmvopt$)
  if $rmvopt = RAND$ **then**
    $lower, upper \leftarrow \alpha * |\mathcal{N}|, \beta * |\mathcal{N}|$
  else if $rmvopt = PROXIMITY$ **then**
    $lower, upper \leftarrow \alpha * \sqrt{|\mathcal{N}|}, \beta * \sqrt{|\mathcal{N}|}$
  end if
  $best \leftarrow sol$
  repeat $nofmutations$ times
  $rmv \leftarrow$ random$(lower, upper)$
  if $rmvopt = RAND$ **then**
    $tmp, free \leftarrow$ RmvR$(best, rmv)$
  else if $rmvopt = PROXIMITY$ **then**
    $seeds \leftarrow$ random$(1, upper/10)$
    $tmp, free \leftarrow$ RmvP$(best, rmv, seeds)$
  end if
  $new \leftarrow$ Insert$(tmp, free)$
  if $fitness(new) > fitness(best)$ **then**
    $best \leftarrow new$
  end if
  end repeat
  return $best$
end function

and an exhaustive search is performed to find the best route and the best position within that route for the node in question. The objective is to minimize the cost of the worst (most costly) route in the solution. The current solution is updated accordingly. The process is repeated until all nodes have been added, and the resulting solution is returned.

The routes are considered in such an order so that the worst route with the largest cost will be checked last. This way, the worst route will be checked only if the node's insertion at any other route makes that route even more costly than the currently worst route. Also, if several insertion options result to the same worst-case cost, as a tie-break we pick the one that minimizes the cost increase for the route where the node is added. These optimizations are not shown in Algorithm 3, for brevity.

Figure 2.1 gives an indicative example for the insertion of a node in a solution that includes two routes (for two salesmen/vehicles). In this case, the node is added to the blue route (on the right) because this does not increase the cost of the worst (most costly) orange route (on the left). Also, the node is added in the blue route in a position that minimizes the cost increase.

The node insertion function is invoked in two places. On the one hand, it is used in the

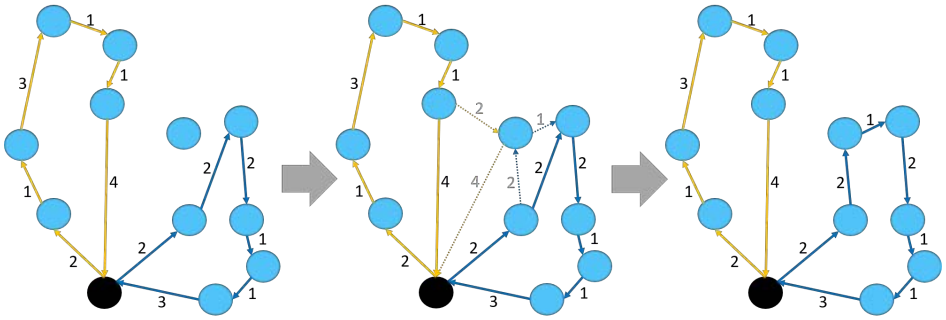


Figure 2.1: Sequence of node insertion for a solution with two routes (from left to right). The algorithm first checks all possible insertion points in both routes (to avoid clutter, only the best option for each route is shown). The algorithm chooses to insert the node in the blue route. This is because the orange route has a total cost of 12 which would further increase to 14 if the node would be added there, whereas by adding the node to the blue route its cost increases to 12, without increasing the cost of the worst / most expensive route, which remains 12 as before.

Algorithm 3 Node insertion method

```
function INSERT(sol, nodes)
    cur ← sol
    while nodes ≠ ∅ do
        minwcost ← ∞                                  ▷ min cost of worst route
        n_j ←rmvNodeRandom(nodes)
        for each r ∈ cur (increasing cost order) do
            for each n_i ∈ r do
                r' ← addNode(r, n_i, n_j)
                wc ← worstCost(cur − r + r')
                if wc < minwcost then
                    bestr, bestr' ← r, r'
                    minwcost ← wc
                end if
            end for
        end for
        cur ← cur − bestr + bestr'
    end while
    return cur
end function
```

top-level TS-LNS() function to construct the initial population. In this case, different random solutions are generated by inserting each time the full set of nodes to an empty solution, thereby building a solution from scratch. On the other hand, it is used in the LNS() function in order to repair a solution, by adding-back the nodes that have been previously removed in the destruction process.

2.4.4 Node removal (route destruction)

For the destruction of a given solution, we support two different node removal methods, which are described in Algorithm 4.

Algorithm 4 Node removal methods

```
function RmvR(sol, nofnodes)
    cur ← sol
    nodes ← pickRandom(N, nofnodes)
    free ← ∅
    for each n ∈ nodes do
        r ← routeOf(cur, n)
        r ← rmvNode(r, n)
        free ← free + n
    end for
    return cur, free
end function

function RmvP(sol, nofnodes, nofseeds)
    cur ← sol
    nofnodes' ← nofnodes/nofseeds
    seeds ← pickRandom(N, nofseeds)
    free ← ∅
    for each s ∈ seeds do
        repeat nofnodes' times
        n ← nearestNode(s)                          ▷ incl. s itself
        r ← routeOf(cur, n)
        r ← rmvNode(r, n)
        free ← free + n
        end repeat
    end for
    return cur, free
end function
```

The first method, shown in function RmvR(), removes a number of nodes from the given solution in a random way. Figure 2.2 gives an example of such random node removal, followed by the node insertion. This method is suitable for the general form of the problem, where edge costs do not necessarily reflect the Euclidean distance between the nodes.

The second method, in function RmvP(), removes nodes in a more targeted way, assuming that the edge costs reflect the Euclidean distance between nodes. The rationale is to remove nodes that are in the proximity of so-called seed nodes (the number of seeds is an additional parameter of this method). The seed nodes are picked randomly, but the rest of the nodes to be removed are picked with reference to the seed nodes. More specifically, for each seed the

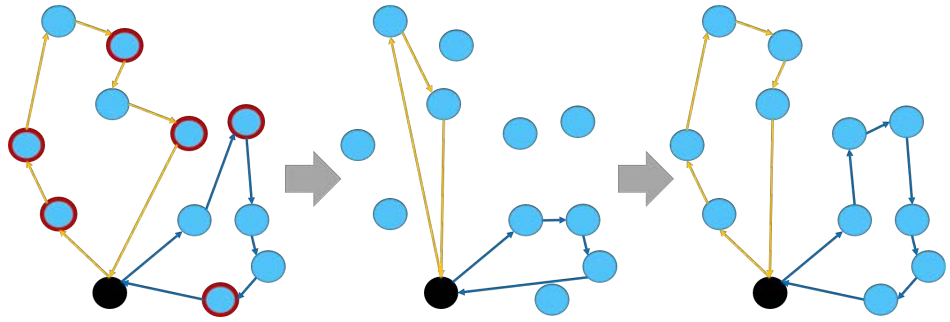


Figure 2.2: Sequence of random destruction with a following repair for a solution with two routes (from left to right). Destruction is performed by removing a total of six nodes. The nodes are all chosen randomly, and then they are reinserted in the solution using the node insertion method (insertion details not shown).

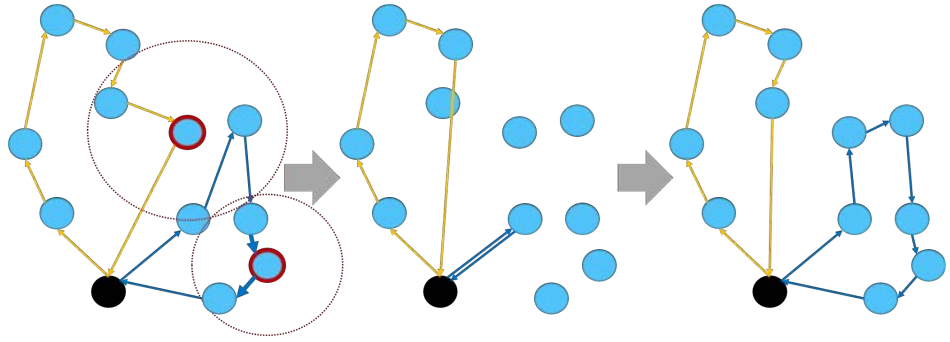


Figure 2.3: Sequence of proximity-based destruction with a following repair for a solution with two routes (from left to right). Destruction is performed based on two randomly selected seed nodes, which happen to belong to different routes. For each seed, another two nodes are removed, chosen based on their proximity to the respective seed node. Note that the nodes are chosen based on their physical proximity to the seed, and may be part of a different route. In total, six nodes are removed, and are reinserted in the solution using the node insertion method (insertion details not shown).

method removes the nodes that are closer to it, based on the costs of the edges that connect the seed to other nodes. As a form of balancing, the total number of nodes to be removed is evenly distributed among the seed nodes. We refer to this method as the proximity-based method, as opposed to the fully random node removal method. Figure 2.3 gives an example of the proximity-based node removal, followed by node insertion.

The node removal functions are invoked from the LNS() function, for a randomly chosen number of nodes. When using the proximity-based node removal method, the number of seed nodes is also chosen in random but from a smaller interval so that the number of seeds is guaranteed to be smaller than the total number of nodes to be removed. Note that RmvR() is always invoked for a number of nodes that is in the order of $|\mathcal{N}|$, whereas RmvP() is invoked for a number of nodes in the order of $\sqrt{|\mathcal{N}|}$. The rationale for removing (and then

reinserting) fewer nodes when using the proximity-based method is that in this case removal is more targeted, around relatively few seed nodes, thus removing a large number of nodes in the same neighborhood would lead to an overly aggressive destruction of the current solution, which actually reduces the chances of finding a better one.

2.4.5 Complexity

We discuss the complexity of the algorithm in a bottom-up fashion, starting from the node insertion and removal functions, then for the large neighborhood search and finally for the entire algorithm. For convenience, we let $N = |\mathcal{N}|$.

The INSERT() function checks for every node to be added every possible insertion point in the routes of the current solution. Given that an exhaustive search is performed for each node, this procedure has complexity of $O(k \times N)$, where k is the number of nodes that need to be added.

The random node removal function RMVR() has $O(k)$ complexity, where k is the number of nodes to remove. The same holds for the proximity-based removal function RMVP(), where k is the total number of nodes to be removed (the seeds plus the nodes in proximity).

In each mutation that is performed within LNS(), the number k of nodes to be removed from and then reinserted into the solution is chosen randomly. However, recall that when using RMVR() then k is in the order of N, but when using RMVP() then k is in the order of $\sqrt{N}$ (see Algorithm 2). As a consequence, in the first case, the combined complexity of every mutation is $O(N) + O(N \times N)$ which translates to $O(N^2)$, whereas in the second case the complexity is $O(\sqrt{N}) + O(N \times \sqrt{N})$ or equivalently $O(N \times \sqrt{N})$.

Finally, we focus on the top-level TS-LNS() function (Algorithm 1). Note that the number of LNS() invocations decrease in each iteration as the size of the population becomes smaller, but the number of mutations that are performed in each invocation of LNS() is also adjusted. Assuming an average of K total LNS mutations in each top-level iteration, and a total number of I iterations, the overall complexity of the algorithm is $O(I \times K \times N^2)$ when using the random node removal method and $O(I \times K \times \sqrt{N} \times N)$ when using the proximity-based method. Note that, in turn, I depends on the size of the initial population $MaxPopSize$, the lower threshold for the population size $MinPopSize$ and the rate f at which the size of population is decreased in each iteration.

2.5 Evaluation of the Offline Algorithm

We compare the proposed TS-LNS algorithm with a state of the art heuristic, the IWO algorithm proposed in [25]. A recent comparison that is presented in [30] shows that IWO is dominant (performs better than other algorithms) in various benchmark problems. Next, we describe the experimental setup and configurations of the two algorithms, and then we discuss the results obtained through experiments on both Euclidean and general problems/graphs.

2.5.1 Setup/configuration

We implement the IWO algorithm and the TS-LNS algorithm in Python 3.5.2, and run them on a Ubuntu 16.04 distribution in a VM using Vmware on top of Windows 10. The machine we use to run the experiments has an Intel i7-8550u CPU at 1.8GHz-4.0GHz and 8GB of RAM. The CPU has 4 physical cores with hyperthreading support for a total of 8 threads. The VM is configured with 6 virtual cores (mapped to 6 threads) and 4GB of RAM.

We configure the IWO algorithm to perform 100 top-level iterations. Each such iteration leads to 300 node removal/insertion operations, yielding a total of 30000 operations.

The TS-LNS algorithm is configured to run for $MaxPopSize = 100$ and $MinPopSize = 6$. The rate of population reduction is set to $f = 2$, so in every iteration we keep only half of the population, the fittest 50% of the solutions. In this configuration, TS-LNS performs four top-level iterations.

Regarding the number of LNS mutations that are performed on each solution of the current population, we initially start with 200 LNS mutations, increasing this number by 200 in each iteration. The rationale is for the search effort to be smaller when the number of solutions is large, and increase as the number of solutions gets smaller. More specifically, 200 mutations are performed for each of the fittest 50 random solutions in the first iteration, 400 LNS mutations are performed for each of the fittest 25 solutions in the second iteration, and 600 LNS mutations are performed for each of the fittest 12 solutions in the third iteration. For the remaining 6 solutions, as an exception, only 467 LNS mutations are performed in order to have a total of 30002 node removal/insertion operations, on par with the IWO algorithm.

We refer to TS-LNS with the random node removal method as TS-LNS-g given that this configuration is more suitable for the general form of mTSP. In this case, we set $\alpha = 0.2$ and $\beta = 0.4$, so that the interval that is used to randomly pick the number of nodes to be

Table 2.1: Results of IWO.

Benchmark	M	Cost Avg	Cost StD	Exec (s)
$eil51$	3	159.56	0	16.58
	5	118.13	0	18.71
	10	112.08	0	22.86
$kroB100$	3	8503.41	22.73	56.69
	5	7008.01	15.06	63.61
	10	6700.04	0	75.57
$ch150$	3	2455.21	20.66	124.20
	5	1768.66	8.86	135.26
	10	1554.64	0	162.98
$lin318$	3	17138.27	161.83	593.22
	5	12379.09	68.36	651.16
	10	9816.99	20.62	751.88

removed is $[0.2 * N..0.4 * N]$. TS-LNS with the proximity-based node removal method is referred to as TS-LNS-e as this is more suitable for the Euclidean form of mTSP. When using this configuration, we set $\alpha = 1.0$ and $\beta = 4.0$, so the respective interval is $[\sqrt{N}..4 * \sqrt{N}]$.

2.5.2 TS-LNS-g vs. IWO for Euclidean problems

In a first set of experiments, we compare TS-LNS-g against IWO. As input graphs, we use the $eil51$, $kroB100$, $ch150$ and $lin318$ benchmarks from the TSPLIB suite [62]. These correspond to Euclidean problems for graphs with $51, 100, 150$ and 318 nodes, respectively. For each benchmark, we run the algorithms for a team of 3, 5 and 10 salesmen. The results for IWO are given in Table 2.1 and for TS-LNS-g in Table 2.2. We report the averages over 20 runs.

As far as the quality of the solutions is concerned, TS-LNS-g produces the same solutions as IWO for the small problem with 51 nodes, and equal or better solutions for the larger problems. More specifically, the solutions of TS-LNS-g are on average marginally better, by 0.1% and 0.2%, than those of IWO for 100 and 150 nodes, respectively. For the problem with 318 nodes, the solution of TS-LNS-g is on average 3.2% better than IWO. The standard deviation is small in all cases, with TS-LNS-g having an even smaller deviation than IWO.

We note that both algorithms manage to find the optimal solution in the problems with 51, 100 and 150 nodes with 10 salesmen. In all these cases, the cost of the solution is indeed equal to twice the cost of the edge that connects the depot node and the node that is farthest away from it (it is impossible for the worst route to have a lower cost). Moreover, TS-LNS-g

Table 2.2: Results of TS-LNS-g.

Benchmark	M	Cost Avg	Cost StD	Exec (s)
$eil51$	3	159.56	0	13.39
	5	118.13	0	15.03
	10	112.08	0	18.13
$kroB100$	3	8497.79	19.69	43.98
	5	6982.58	17.05	48.02
	10	6700.04	0	59.40
$ch150$	3	2446.41	15.03	97.41
	5	1764.80	8.68	107.05
	10	1554.64	0	128.31
$lin318$	3	16556.07	109.40	451.50
	5	11701.93	53.67	486.72
	10	9731.16	0	581.72

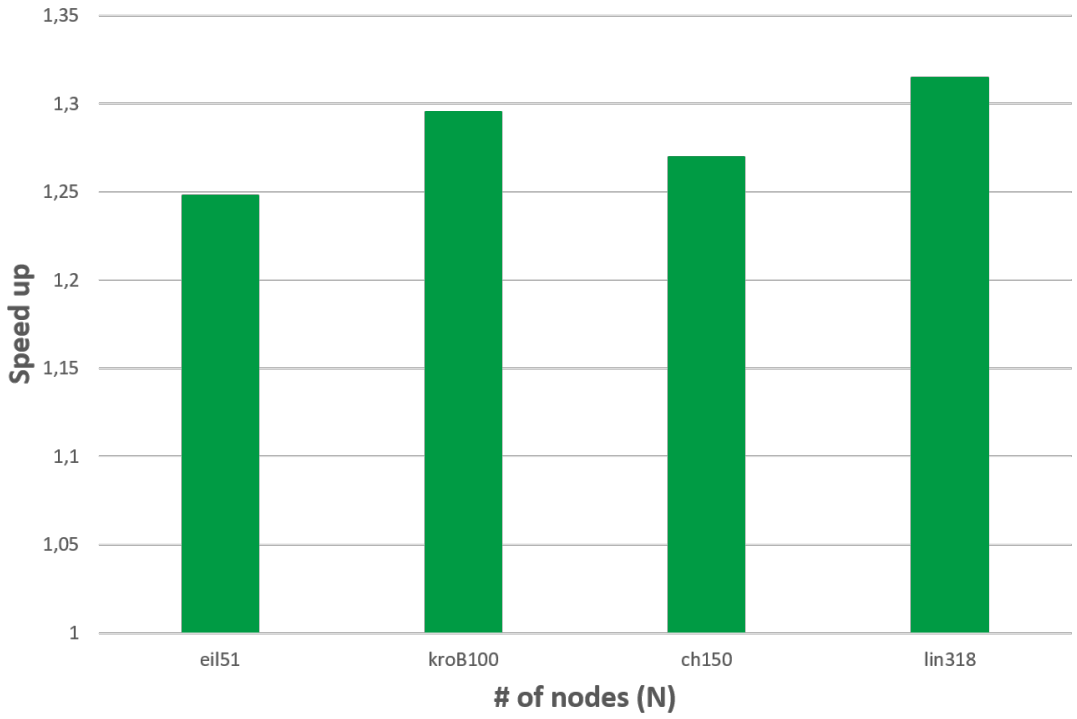


Figure 2.4: Speed-up of TS-LNS-g vs. IWO.

also finds optimal solution for the problem with 318 nodes and 10 salesmen.

At the same time, TS-LNS-g is considerably faster than IWO, as shown in Figure 2.4. The average speed-up is equal to $1.25x$, $1.30x$, $1.27x$ and $1.31x$ for the benchmarks with 51, 100, 150 and 318 nodes, respectively, at an overall average of $1.28x$. Note that both algorithms perform the same number of mutations, with each mutation (node removal and reinsertion operation) having $O(N^2)$ complexity. However, the mutations of TS-LNS-g involve fewer nodes, on average $0.3 \times N$ vs. $0.5 \times N$ in IWO, leading to a smaller total number of node

Table 2.3: Results of TS-LNS-e.

Benchmark	M	Cost Avg	Cost StD	Exec (s)
$eil51$	3	159.56	0	13.98
	5	118.13	0	15.87
	10	112.08	0	19.41
$kroB100$	3	8482.50	5.88	34.43
	5	6965.85	17.56	38.81
	10	6700.04	0	46.98
$ch150$	3	2416.55	13.47	65.09
	5	1747.36	5.66	72.32
	10	1554.64	0	86.62
$lin318$	3	16113.78	46.12	215.62
	5	11500.98	43.78	236.42
	10	9731.16	0	281.31

removal/reinsertions. This reduction in the search space does not seem to have any impact on the solution quality of TS-LNS-g.

2.5.3 TS-LNS-g vs.TS-LNS-e for Euclidean problems

In a second series of experiments, we run TS-LNS-e for the same same set of benchmarks as above. Recall that TS-LNS-e is designed to work well specifically for Euclidean problems. Table 2.3 shows the results. Again, the averages over 20 runs are reported.

We observe that TS-LNS-e produces the same results as TS-LNS-g and IWO for the problems with 51 nodes, and finds better solutions for all larger problems. More specifically, for the problems with 100, 150 and 318 nodes, the solutions of TS-LNS-e are on average 0.1%, 0.7% and 1.5% better than TS-LNS-g, and roughly 0.3%, 0.9% and 4.6% than the solutions found by IWO.

The standard deviation of TS-LNS-e is less or equal to that of TS-LNS-g in most of the problems. As an exception, for 100 nodes and 5 salesmen/vehicles the deviation of TS-LNS-e is slightly larger than TS-LNS-g for the same problem, but it is also higher than that of TS-LNS-e itself for the problem with 100 nodes and 3 salesmen/vehicles. This could be an indication that it might be beneficial to take into account the number of salesmen/vehicles when deciding the number of nodes to be removed/reinserted in each mutation.

Importantly, TS-LNS-e is also much faster than TS-LNS-g for bigger problem sizes, as shown in Figure 2.5. The average speed-up is $1.26x$, $1.49x$ and $2.07x$, for the benchmarks with 100, 150 and 318 nodes, respectively. This performance is even more impressive if com-

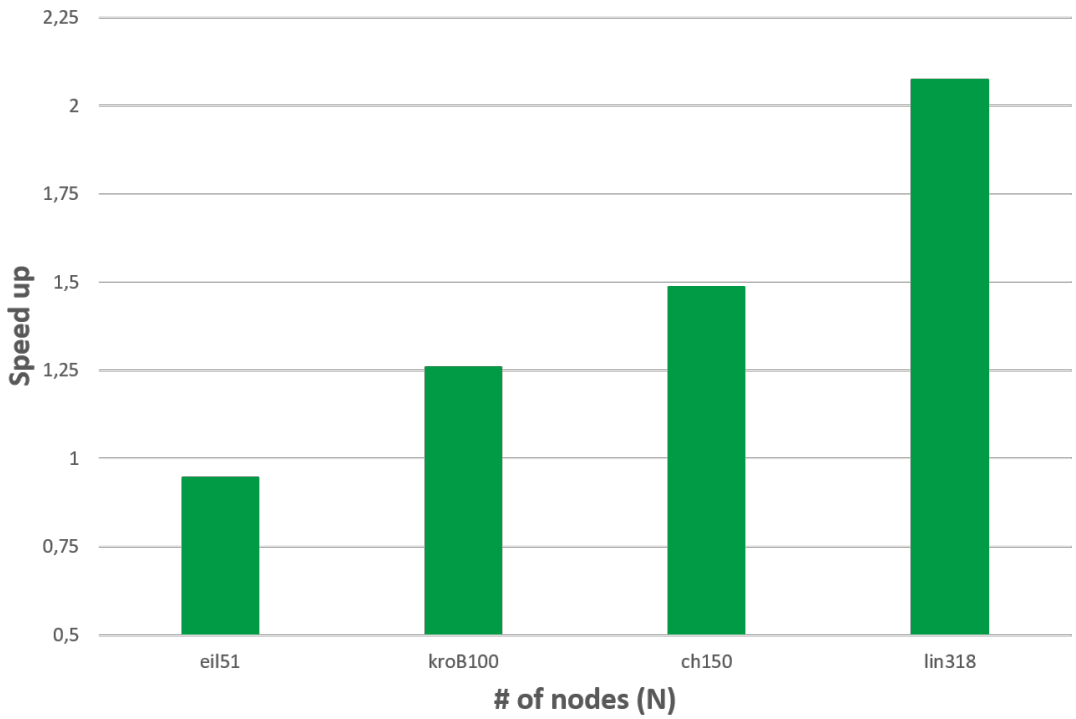


Figure 2.5: Speed-up of TS-LNS-e vs. TS-LNS-g.

pared with IWO, yielding a speed-up of $1.64x$, $1.89x$ and $2.71x$, for these benchmarks. This significant improvement is due to the lower $O(N \times \sqrt{N})$ complexity of TS-LNS-e compared to $O(N^2)$ for TS-LNS-g and IWO.

Note, however, that TS-LNS-e is somewhat slower than TS-LNS-g for the smallest problem with 51 nodes. The reason is that, in this particular case, the interval $[\sqrt{N}..4 * \sqrt{N}]$ used in TS-LNS-e to decide the number of nodes to be removed/reinserted in each mutation, becomes $[7..29]$ and has a much larger upper bound than the interval $[0.2 * N..0.4 * N]$ used in TS-LNS-g, which is $[10..20]$. As a result, in each mutation, TS-LNS-e removes/reinserts on average a larger number of nodes than TS-LNS-g.

2.5.4 TS-LNS-g/e vs. IWO for general problems

In a last series of experiments, we evaluate the algorithms using as input more general graphs. For this purpose we use three different benchmarks of the TSPLIB suite [62], $kro124p$, $gr120$ and $ftv170$ with 100, 120 and 171 nodes, respectively. In all cases, the edge costs are non-Euclidean. In $gr120$ the edge/travel costs are symmetrical, whereas in $kro124p$ and $ftv170$ they are asymmetrical. In order for TS-LNS-e to work on graphs with asymmetrical costs, the nodes to be removed around a seed are chosen based on the two-way trip cost

Table 2.4: Solution cost of the algorithms for general graphs.

Benchmark	M	IWO	TS-LNS-g	TS-LNS-e
$kro124p$	3	13470.05	13539.90	13313.2
	5	9137.3	9157.15	8990.55
	10	6419.4	6343.45	6322.45
$gr120$	3	2614.15	2604.95	2580.50
	5	1834.25	1823.0	1812.30
	10	1558.0	1554.40	1555.35
$ftv170$	3	1026.75	1026.1	986.65
	5	688.85	673.63	654.15
	10	447.70	432.37	427.53

between them.

Table 2.4 reports the cost of the solutions that are generated by each algorithm, averaged over 20 runs. As in the previous experiments, the standard deviation is small. The previous observations regarding the execution speed of the algorithm also hold for these experiments (not reported in more detail here, for brevity).

It can be seen that both TS-LNS variants once again produce solutions that are close and most of the times even better than those of IWO, yielding an improvement of up to 3.4% for TS-LNS-g or even 5.0% for TS-LNS-e. In fact, TS-LNS-e always generates better solutions than IWO. We attribute this (somewhat surprisingly) good performance to the fact that these general benchmark graphs are non-random and specifically in the case of gr120 and ftv170 they are based on real-world routes and travel costs. So, even though the costs are not a direct function of the straight-line Euclidean distances, the most significant costs components that mostly affect the decisions that are taken by the algorithms probably still have a strong affinity to this.

2.6 Online Algorithm

In this section we propose a heuristic that starts from an initial plan and then adjusts it during the mission time performing a large neighborhood search (LNS). Notably, in this case, the initial paths can be planned based on *optimistic estimates* about the travel costs, where the estimation can be smaller than the worst case cost $c_{i,j}^{max}$. If the actual costs that are experienced during mission execution turn out to be even lower, an attempt is made to find a better plan. Else, if a path turns out to be risky based on the current energy reserves, a detour is introduced

back to the depot. In the following, we explain the most important elements of the heuristic and provide an algorithmic description for it.

2.6.1 Initial plan

The TS-LNS algorithm can be adapted in a straightforward way for vehicles that have capacity/energy constraints and need to reload/refuel at a depot node. More specifically, the feasibility of a node insertion must be checked before adding the node to a path. The insertion is considered as a candidate only if it does not exceed the current capacity (remaining energy) of the corresponding vehicle. If a node cannot be inserted in any of the M paths, a new path is created for visiting that node, and this is assigned to the vehicle with the smaller makespan. As a result, each vehicle m may have more than one paths assigned to it, and each entry s[m] in the plan contains a list of paths (as opposed to just one path). A new path is always added at the end of the vehicle's path list.

2.6.2 Plan representation and state information

The plan (schedule) is internally captured using M tuples, one for each vehicle, of the form $\langle paths[], rem_{est}[], curr, rem_{curr}, pos, rem_{pos}\rangle$. The $paths[]$ field is the list of paths assigned to the vehicle (as discussed in Section 2.3.3), sorted in descending order with respect to their energy costs. Also, $rem_{est}[k]$ is the remaining energy of the vehicle at the end of $paths[k]$ as estimated in the planning phase. This is equal to $rem(B, paths[k])$ since each path starts from the depot.

Each vehicle follows the paths in the order these appear in its $paths[]$ list. The index of the path that is currently being followed is kept in $curr$. The estimate for the energy that should remain available at the end of the current path is updated after each hop, and is kept in rem_{curr}. This is used to detect major deviations vs. the initial estimate $rem_{est}[curr]$. Finally, pos records the position of the vehicle in the current path, while rem_{pos} is the remaining energy of the vehicle at that position. Each time the vehicle begins to pursue a new path, $pos = 1$. Also, $rem_{pos} = B$ since every path starts from the depot node n_d.

2.6.3 Cost estimation

Path construction is done so as to minimize the makespan (Section 2.3.3) while ensuring feasibility (Section 2.3.4). Since the actual travel costs are unknown, feasibility checks are based on estimations about the energy that will be required by the vehicle to perform the planned hops.

More specifically, the algorithm uses a plug-in function $est()$ that provides estimates for the travel costs. This function is configurable to be able to experiment with different estimation methods without modifying the algorithm. The most conservative approach is to let $est(c_{i,j}) = c_{i,j}^{max}$, pessimistically assuming that each hop will incur the maximum possible energy cost. In this work, we explore more optimistic estimates which are lower than the worst-case costs.

Notably, the algorithm is designed to be fully safe. In other words, it guarantees that all vehicles are able to return to the depot node, even in the worst-case scenario where the travel costs are the largest possible. This safety is achieved in an elaborate way, making it possible to find better plans compared to fully conservative planning.

2.6.4 Core logic

Given an initial plan, each vehicle starts with the first path in its list. Thus, $curr = 1$, $rem_{curr} = rem_{est}[curr]$, $pos = 1$ and $rem_{pos} = B$ (every path starts from the depot). Then, as long as some vehicles are still active, their status is tracked and the plan is adapted accordingly. The main control loop is shown in Algorithm 5, while the core logic of the heuristic is given in Algorithm 6.

Algorithm 5 Online path planning algorithm.

```
function PLAN(s, V, n_d)
    while ∃m : s[m].curr ≤ |s[m].paths| do
        replan ← false
        for each m that performs its next hop do
            UPDATEVEHICLESTATE(s[m])
        end for
        if replan then LNS(s, nofmutations, rmvopt)
        end if
    end while
end function
```

After each hop, pos is incremented, rem_{pos} is set to the remaining energy at this position,

and rem_{curr} is updated to contain the new estimate for the remaining energy at the end of the current path, based on the actual costs for the hops that have been performed so far. It is then checked whether, based on rem_{pos}, the vehicle can afford to make the next hop, from node $n_j = paths[curr][pos]$ to $n_k = paths[curr][pos+1]$, and still be able to return to the depot n_d, even under worst-case travel costs. This safety check is expressed as

$$rem_{pos} - c_{j,k}^{max} - c_{k,d}^{max} > 0 \tag{2.3}$$

If Equation 2.3 does not hold, a so-called detour is introduced so that the vehicle returns to the depot, and the rest of the path is added as a new path starting from the depot. Also, a replan attempt is made just in case the vehicle may visit some more nodes of interest on its way back to the depot.

If Equation 2.3 holds, it is checked whether the updated estimate for the remaining energy at the end of the current path is significantly higher than the initial estimate

$$\frac{rem_{curr} - rem_{est}[curr]}{rem_{est}[curr]} \geq Dev_{replan} \tag{2.4}$$

in which case, a replan attempt is made. The rationale is to exploit the extra available energy to find a better plan. The replanning threshold Dev_{replan} is configurable. Note that if the cost estimation is pessimistic, assuming the maximum travel costs, it is guaranteed that Equation 2.3 will always hold. As a consequence, no detours will ever be produced in this case.

Finally, a replanning attempt is made when a vehicle completes the paths assigned to it. This is to correct any imbalance by letting the free vehicle take over some of the nodes assigned to other vehicles, which can further reduce the makespan.

Algorithm 6 Main functions.

function UPDATEVEHICLESTATE($s[m]$)

if $pos = |paths[curr]|$ **then** ▷ was last hop

if $curr < |paths|$ **then** ▷ not done

$curr \leftarrow curr + 1$ ▷ start next path

$pos, rem_{pos} \leftarrow 1, B$ ▷ restore at the depot

$rem_{curr} = rem_{est}[curr]$

else

$replan \leftarrow true$ ▷ exploit free vehicle

end if

else

$n_i \leftarrow paths[curr][pos]$

$n_j \leftarrow paths[curr][pos + 1]$

$pos \leftarrow pos + 1$

$rem_{pos} \leftarrow rem_{pos} - c_{i,j}$

$rem_{curr} \leftarrow rem_{curr} + (est(c_{i,j}) - c_{i,j})$

$n_k \leftarrow paths[curr][pos + 1]$

if $rem_{pos} - c^{max}_{j,k} - c^{max}_{k,d} \leq 0$ **then**

ADDDETOUR($paths, curr, pos$)

$replan \leftarrow true$ ▷ exploit return to depot

else if $\frac{rem_{curr} - rem_{est}[curr]}{rem_{est}[curr]} \geq Dev_{replan}$ **then**

$replan \leftarrow true$ ▷ exploit extra energy

end if

end if

end function

function ADDDETOUR($paths, curr, pos$)

$p = paths[curr]$

$p1 \leftarrow p[1 : pos] + n_d$ ▷ detour to depot

$p2 \leftarrow n_d + p[pos + 1 : |p|]$ ▷ remaining path

replacePath($paths, curr, p1$)

insertPath($paths, p2$)

end function

2.6.5 Path replanning

The replanning step is performed by running the algorithm described in Section 2.4. Since the problem is Euclidean, we choose to use the proximity-based node removal method (RmvP function described in Algorithm 4).

The feasibility of the constructed paths is checked based on the estimated travel costs $est(c_{i,j})$. In addition, the safety check as per Equation 2.3 is performed before changing the next hop in the current path of any vehicle, ensuring that the vehicle will be able to return to the depot even under the worst-case travel costs. If a node can not be inserted in any path under these constraints, a new path is formed, with that node being the only node in the path. This path is assigned to a vehicle so that the plan's makespan is minimized.

2.7 Dynamic Path Adaptation Evaluation

The evaluation of the online algorithm is two-fold. First we evaluate our approach when the paths are planned and adapted pessimistically, based on the worst case costs. In this case a new depot detour will never be required, as the actual costs experienced will always be lower than expected (rendering the paths always feasible). In the respective experiments, we vary the depot placement, the threshold for the replanning and the search intensity. Then we evaluate our approach with more optimistic cost estimations keeping the replanning threshold and the search intensity fixed, while varying the depot placement, the fleet size and degree of optimism. All experiments are conducted via simulations.

2.7.1 Pessimistic cost estimation

Experimental setup

We conduct our experiments for a 11×11 grid of 121 nodes. The nodes represent the geographical locations of an area that has to be scanned by a team of vehicles exhaustively, by visiting all nodes. We assume that the vehicles can freely move between nodes in a straight line. This is typically the case for aerial unmanned vehicles (UAVs) that scan a large area from a relatively high altitude that keeps them safely above trees and power lines. Thus, there is an edge between every two nodes.

The cost $c_{i,j}$ of each edge $e_{i,j}$ represents the cost for moving from n_i to n_j. This cost is

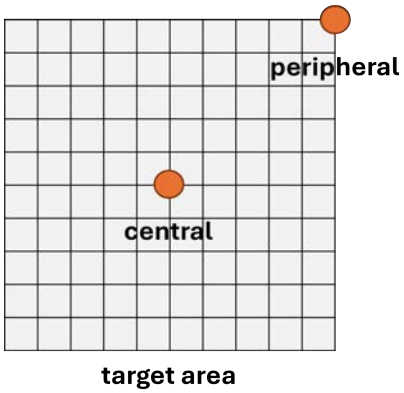


Figure 2.6: Depot placements with respect to the target area.

not known with certainty, but randomly varies following a uniform distribution $[c_{i,j}^{min}..c_{i,j}^{max}]$. The upper and lower bounds of the cost distribution are a function of the Euclidean distance between the nodes' positions, let $euc_{i,j}$. More specifically, $c_{i,j}^{min} = \alpha \times euc_{i,j}$ and $c_{i,j}^{max} = \beta \times euc_{i,j}$. We investigate scenarios for *small uncertainty* ($\alpha = 0.5$ and $\beta = 1$) and *large uncertainty* ($\alpha = 0.25$ and $\beta = 1$). Without loss of generality, we let the travel time be a linear function of the edge cost.

To capture the uncertainty of the travel cost, we create 50 different scenarios. In each scenario, the cost of every edge is randomly chosen based on the respective random distribution. The actual cost of an edge is discovered only at runtime, when a vehicle crosses that edge and reaches the destination node. Given that each node has to be visited once and that the graph is fully connected, every edge is traversed at most once, thus it suffices to have only one randomly chosen cost value for each edge.

In our experiments we assume a single depot node, but investigate two different scenarios regarding its location. In the *peripheral depot* scenario the depot node is located at one of the corners of the grid, while in the *central depot* scenario the depot node is located at the center of the grid, as shown in Figure 2.6.

We use a fleet of $M = 3$ vehicles, which all have the same energy capacity. The capacity of the vehicles is set to the worst-case (maximum) round-trip cost between the depot node and the node that is farthest away from it. This ensures that it is indeed possible to visit all nodes, even if the edge costs turn out to be the largest possible. Note that B has a larger value in the scenario where the depot is further away from the center of the target area.

Configurations of the online algorithm and reference

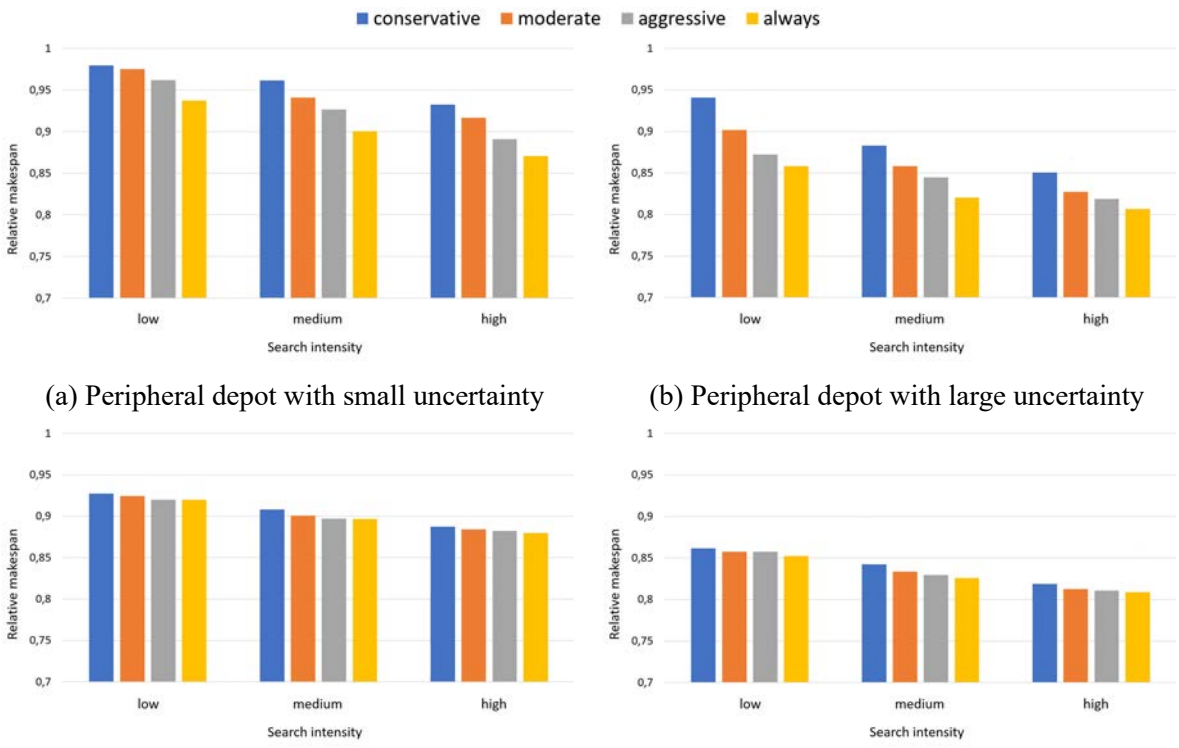
We test the online algorithm for various configurations. On the one hand, we experiment with four replanning thresholds 0.2, 0.1, 0.05 and 0.0, referred to as *conservative*, *moderate*, *aggressive* and *always* configurations, respectively. Recall that lower thresholds lead to more frequent/earlier replanning. In the *always* configuration the algorithm replans whenever a deviation is experienced, irrespectively of how significant this is relative to the path cost. On the other hand, we vary the number of iterations in the LNS component, from $25, 50$ to 100 iterations. We refer to this as *low*, *medium* and *high* search intensity, respectively. Path changes do not change the current destination of a vehicle but may affect the remaining path(s). The algorithm executes quickly (under a second) thus it can run even after every hop without delaying the vehicles.

As a reference, we use a plan that is produced offline. For this we use the algorithm described in Section 2.4 (with the extensions described in Section 2.6.1) which provides good results for the min-max mTSP problem. The offline plan is generated based on the worst-case cost of each edge, and thus is guaranteed to be feasible (no vehicle will ever run out of energy) independently of the actual costs that will be experienced by the vehicles during the mission. The offline plans are also used as the initial plans for the online algorithm.

Results

We run each configuration of the online algorithm 5 times for each of the 50 edge cost scenarios (for each depot and uncertainty scenario). Figure 2.7 shows the average makespan of the different configurations as a percentage of the makespan of the offline algorithm. At the top, Figure 2.7a and Figure 2.7b show the results for the peripheral depot scenario, while Figure 2.7c and Figure 2.7d at the bottom show the central depot scenario. The figures on the left show the results for the small uncertainty scenario, and the ones on the right for the large uncertainty scenario. For the two "extremes" of our experimental study, the improvement varies from 2% for the most conservative replanning and low search intensity configuration in the peripheral depot node scenario with small uncertainty, up to nearly 20% for the most aggressive replanning and highest search intensity configuration in the central depot node scenario with large uncertainty.

Increasingly better results are achieved when the online algorithm replans more aggressively (more often). This is because the sooner one replans during the mission, the more



(a) Peripheral depot with small uncertainty

(b) Peripheral depot with large uncertainty

(c) Central depot with small uncertainty

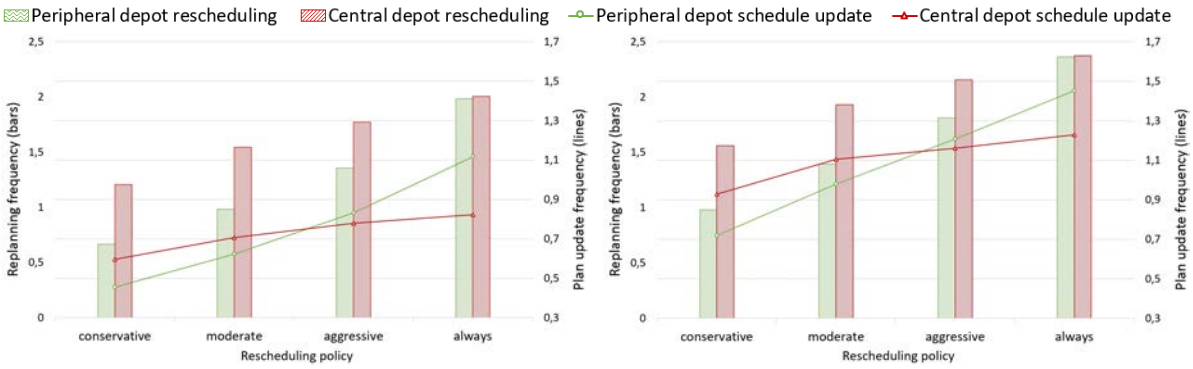
(d) Central depot with large uncertainty

Figure 2.7: Makespan of the online algorithm vs. the offline algorithm.

likely it is to achieve a significant optimization in the routes of the vehicles as there are still many unvisited nodes. More conservative (less frequent) replanning misses such opportunities. Even though the accumulated reserves are larger in this case, replanning takes place after having visited several nodes, so there are fewer opportunities for major optimizations. Also, as expected, better results are achieved for the more intensive search configurations.

The plan improvements are better under larger cost uncertainty. Given that path planning is based on worst-case estimates, any planning decisions become increasingly sub-optimal under larger uncertainty, and this can only be repaired by replanning more often.

Note that the improvements are more significant in the central depot scenario for the conservative, moderate and aggressive replan policies. The reason is that the paths are shorter so the respective worst-case estimates are smaller than in the peripheral scenario. As a consequence, the same absolute cost deviations meet the respective thresholds more often, leading to more frequent replanning attempts. The always replan policy achieves an equally good improvement in both depot scenarios. In this case, a replanning attempt is done at the slightest cost deviation (even if minor compared to the estimated path costs), thus replanning is performed equally frequently in both scenarios.



(a) Small uncertainty (b) Large uncertainty

Figure 2.8: Replanning and plan update frequency of the online algorithm (average over all search intensity configurations).

Figure 2.8 shows the replanning and plan update frequencies, averaged over all search intensity configurations. These are calculated by dividing the total number of replanning attempts and respectively the total number of actual plan updates that were performed during the mission, by the makespan. In practical terms, this corresponds to the average number of replanning attempts or plan updates performed while a vehicle is traveling between two nodes; values can be larger than 1 as there are many vehicles that travel concurrently over different distances.

Naturally, the replanning attempt frequency (bars) increases with more aggressive replanning policies and for larger uncertainty. As discussed above, the conservative, moderate and aggressive policies (non-zero thresholds) lead to more frequent replanning in the central than in the peripheral depot scenarios, while the always replan policy (zero threshold) leads to practically the same replanning frequency.

The more replanning attempts are made, the more likely it is that at least some of them will succeed, leading to a higher update frequency (lines). Note that conservative and moderate replanning leads to a higher update frequency in the central vs. the peripheral depot scenarios, whereas the situation is reversed in the aggressive and always replanning policies. Replanning attempts are in general more likely to succeed in the peripheral depot scenarios because paths are longer hence the worst-case estimates are also more likely to be overly pessimistic. Therefore, as the gap of the replanning frequency between the two scenarios shrinks, the update frequency becomes higher in the peripheral depot scenario. Still, the impact of each individual update is less significant in the peripheral depot scenario, which is the reason why practically the same improvement is achieved in both cases with the always

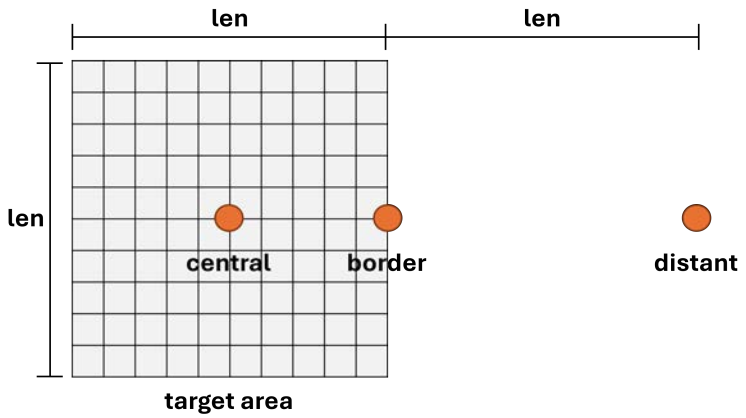


Figure 2.9: Depot placements with respect to the target area.

replan policy.

While the proposed heuristic clearly outperforms the static planning, it is far from optimal. An oracle with a priori knowledge of the costs that will be experienced during the mission, can further reduce the makespan by 25% up to 45% depending on the scenario. So, there seems to be plenty of room for optimizations, provided one is willing to plan riskier paths based on more optimistic cost estimations that may require "emergency" depot detours. The next subsection focuses in this.

2.7.2 Optimistic cost estimation

Topology

In these experiments we use the same grid as a target area as in the previous section. For the depot node, we investigate three placement scenarios, Figure 2.9. In the *central* placement scenario, the depot is at the center of the target area. In the *border* scenario, it is at the borderline of the area. Finally, in the *distant* scenario, the depot is placed further away from the target area, at a distance equal to the size of the edge of the square area.

Energy costs and uncertainty

Similarly to the previous experiments, the edge costs $c_{i,j}$ follow a uniform distribution $[c_{i,j}^{min}..c_{i,j}^{max}]$. Let the Euclidean distance between n_i and n_j be the average of this distribution $c_{i,j}^{avg}$. Again we experiment with two types of uncertainties. In *low* uncertainty, $c_{i,j}^{min} = c_{i,j}^{avg} \times \frac{3}{4}$ and $c_{i,j}^{max} = c_{i,j}^{avg} \times \frac{5}{4}$. In *high* uncertainty, we set $c_{i,j}^{min} = c_{i,j}^{avg} \times \frac{2}{3}$ and $c_{i,j}^{max} = c_{i,j}^{avg} \times \frac{4}{3}$. Without loss of generality, we let travel time be a linear function of edge cost.

To ensure that the problem has a safe solution, we make the same assumptions about the maximum capacity B of the drones as in Section 2.7.1. Note that in this setup, the maximum energy of the drones is larger not only in scenarios that the depot is further away from the center, but also when the uncertainty is higher.

Cost estimation

We test the algorithm for two optimistic variants of the cost estimation function $est(c_{i,j})$. In the *aggressive* variant, the function returns $c_{i,j}^{avg}$, whereas in the *moderate* variant it returns $\frac{c_{i,j}^{avg}+c_{i,j}^{max}}{2}$. Note that the latter estimate is more conservative compared to the former. We run all variants with a replan threshold of $Dev_{replan} = 5\%$.

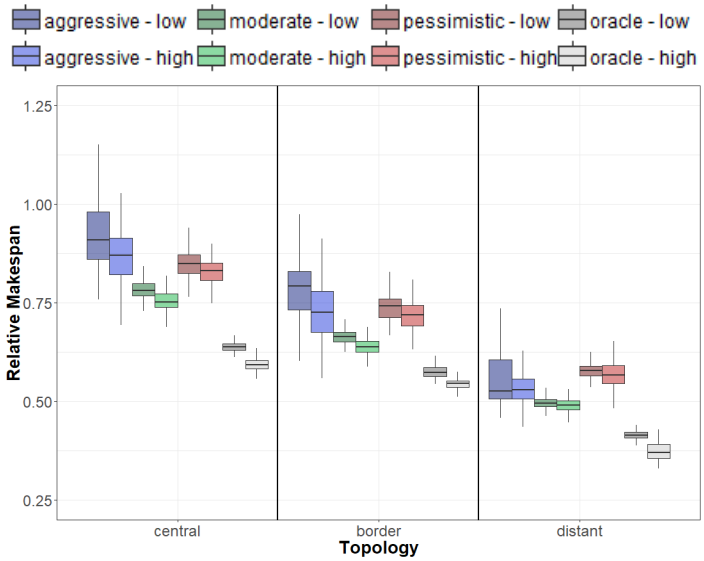
As a reference, we also run the algorithm in the most *pessimistic* variant, where the cost function returns the worst-case cost $c_{i,j}^{max}$ (this is equivalent to the variant evaluated in Section 2.7.1). Moreover, we report the results achieved by an offline *oracle* algorithm that has perfect a priori knowledge of the actual cost of each hop during the mission.

Results

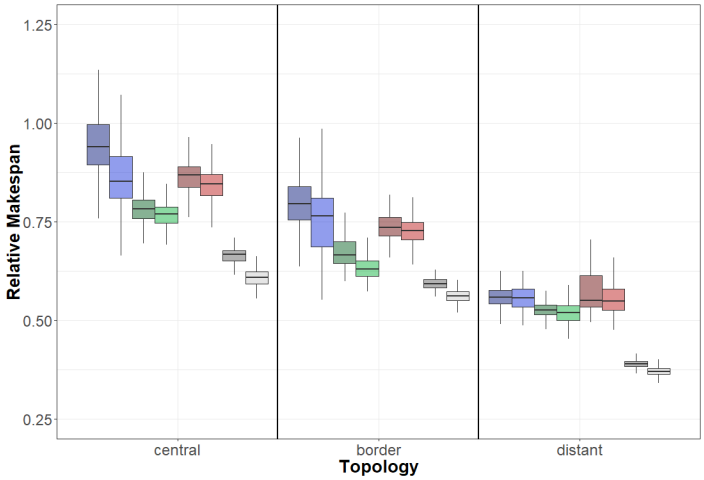
For each depot placement and cost uncertainty scenario, we run the variants of the algorithm for 5 different initial schedules and 50 different randomly generated edge cost settings (based on the respective cost distribution). The initial placements are produced using the algorithm of Section 2.4, while the cost estimation function is the same as in the online algorithm, depending on the variant.

Figure 2.10 shows the relative makespan achieved in each scenario by the different configurations of the algorithm for a fleet of 1, 2 and 3 vehicles. Each boxplot summarizes the results obtained in the 250 experiments (5 initial schedules, each for 50 travel cost settings). To make a fair comparison, between plans that are safe and ensure that the vehicles do not exhaust their energy, the reference for the online variants is the plan produced offline using pessimistic cost estimates. Note that the absolute values are different for each topology and uncertainty. Also, since different uncertainties lead to different actual travel costs, the respective schedules produced by the oracle differ too in each case.

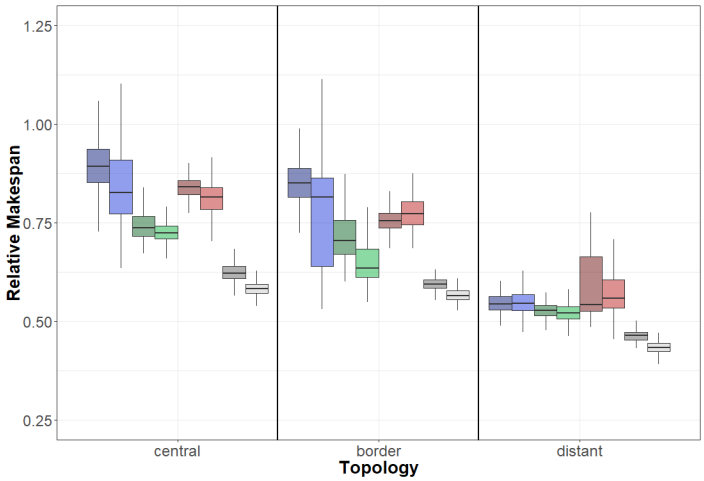
We observe that all online variants achieve increasingly better results as the depot is placed further away from the target area. This is because the offline plan becomes increasingly suboptimal as the worst-case travel costs increase, and thus there is more room for



(a) 1 vehicle



(b) 2 vehicles



(c) 3 vehicles

Figure 2.10: Relative makespan vs. offline for the different depot placements with 1, 2 and 3 vehicles.

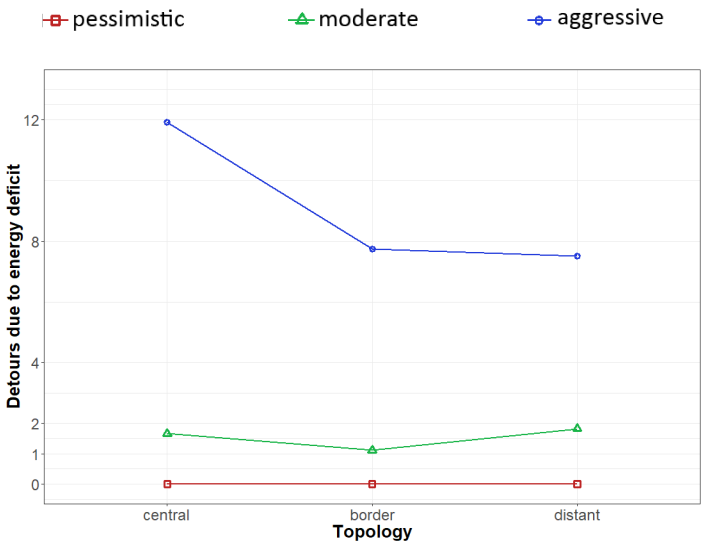
improvement. Similarly, the results achieved for the high uncertainty are generally better than for low uncertainty. Again, the reason is that higher uncertainty increases the worst-case costs thereby creating more room for improvement vs. the offline solution.

Notably, the aggressive variant achieves worse results than the pessimistic variant in the central and border depot scenarios. It manages to achieve consistently equal or slightly better results only in the distant depot scenario. The reason is that it is overly optimistic and often produces paths which turn out to be risky (paths where at some point Equation 2.3 no longer holds). This, in turn, leads to a large number of detours (see Figure 2.11a) which also translate to more trips back to the depot (see Figure 2.11b).

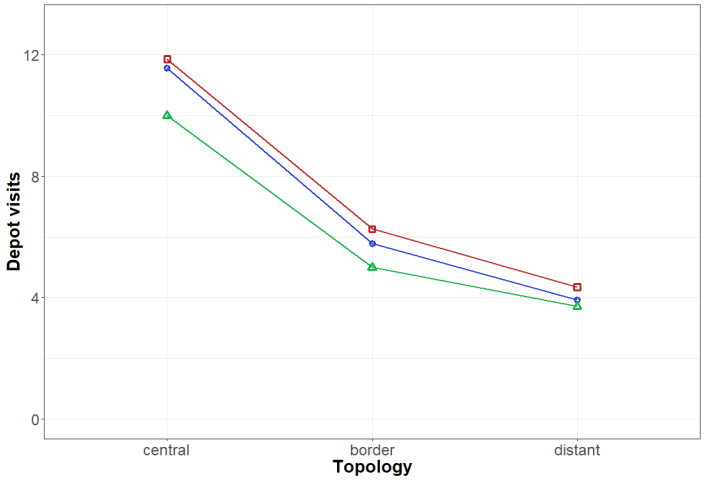
In contrast, the moderate variant outperforms both the pessimistic and the aggressive variant in all cases, reducing the makespan by up to 51% vs. the offline solution and up to 18% vs. the pessimistic variant. This shows that optimistic planning can lead to significantly better results – provided the degree of optimism is not excessive. Note that in the distant depot scenario, the difference with the pessimistic variant becomes quite significant for 1 vehicle, whereas all variants achieve similar results for 2 and 3 vehicles. Also, in the distant depot scenario with 3 vehicles, the results become quite close to the ones of the oracle. This is because a larger number of vehicles decreases the number of assigned paths per vehicle, which also reduces the opportunities for very significant optimizations.

As can be seen in Figure 2.11a, the moderate variant introduces significantly fewer detours than the aggressive variant; in fact, the number of detours remains practically constant across all depot placements. This clearly shows that the paths produced by the moderate variant are more realistic compared to those of the aggressive variant. The aggressive variant introduces significantly fewer detours in the border and distant depot scenarios. In these cases, vehicles have larger energy capacity hence any cost deviations that occur when the vehicle is in the target area become less critical with respect to the safety of the planned paths (it becomes more likely for Equation 2.3 to hold despite cost deviations). Note that the pessimistic variant does not perform any detours because the actual costs cannot possibly be larger than the (worst-case) estimates.

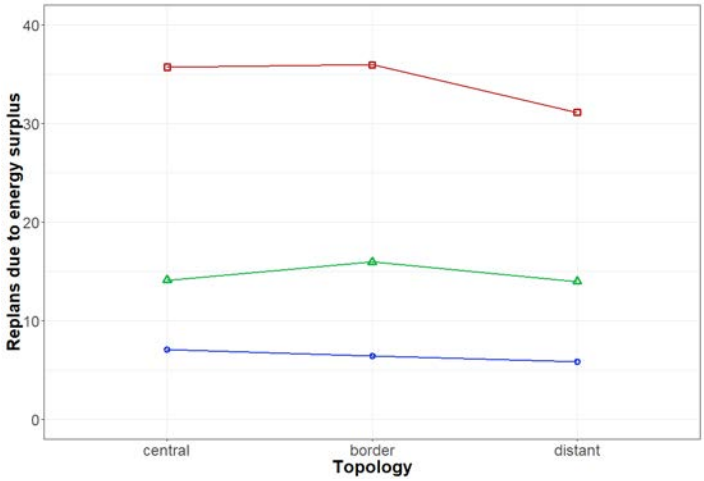
In Figure 2.11b, we report the number of intermediate depot visits in order for the vehicles to restore their energy reserves during the mission. The moderate variant consistently leads to fewer visits than all other variants. This difference is more significant in the central and border depot scenarios, which also explains the notably lower makespan achieved in those



(a) Detours due to energy deficit



(b) Depot visits



(c) Replans due to energy surplus

Figure 2.11: Features of the plans produced for the different depot placements (average over all uncertainty/vehicle scenarios).

cases (see Figure 2.10). Note that the aggressive variant leads to almost as many depot visits as the pessimistic variant. Still, the makespan of the aggressive variant is worse or equal to the pessimistic variant in the central and border depot placement scenarios (see Figure 2.10). We attribute this to the fact that the aggressive variant performs a much larger number of detours. Even though each detour practically also triggers a replan, which may avoid a depot visit (as can be seen by comparing Figure 2.11a with Figure 2.11b), replanning out of need, under the pressure of energy deficit, is apparently more disruptive and leads to lower-quality paths compared to replanning under energy surplus (also compared to the initial path planning that is performed offline).

Figure 2.11c shows the number of replans due to energy surplus. Note that the trend is opposite to that of the detours, namely the number of replans decreases with increasing degree of optimism. The more optimistic the estimate of travel costs, the less likely it is for the actual aggregate cost to be smaller than the estimate, so that the respective deviation becomes not only positive but also larger than the replan threshold. As expected, the pessimistic variant performs the largest number of replans. Since all cost estimates are made based on the maximum possible costs, the actual costs will turn out to be significantly lower with high probability hence Equation 2.4 holds more often compared to other variants.

Finally, it is worth noting that the moderately optimistic variant performs quite well even compared to the oracle. More specifically, the average makespan over all uncertainty, depot placement and vehicle scenarios, is only about $1.2x$ larger than that of the oracle. Nevertheless, this also shows that there is still some room for improvement.

Chapter 3

Joint Path and Offloading Scheduling in the Edge for Multiple Drones

3.1 Introduction

Besides simple data collection tasks where the processing and analysis of this data takes place offline after the mission is completed, there are cases where the drone has to process the data gathered on each point in situ and in a synchronous manner, before it moves to the next point of interest. However, small drones typically carry resource-constrained computing platforms. As a consequence, heavyweight computations can take a long time to complete, which, in turn, can significantly increase the total time it takes for the drone to complete the mission. One way to reduce the mission time is to offload such computations to more powerful nearby edge servers, instead of performing them onboard. However, this is not always feasible when the servers have limited capacity and there are several drones, each one pursuing its own mission independently of others, which are competing for these resources.

To this end, one must plan the path and jointly schedule the offloading decisions of each drone, so that all drones can benefit from the available edge resources. The problem becomes more challenging when the drones are heterogeneous, making it harder to achieve a balanced and fair schedule. Note that the typically limited flight time of drones, especially if they are battery-powered, makes the problem even more challenging. In this case, failure to offload the computation to an edge server as planned, not only increases the mission time but – as a side-effect of the additional time that the drone spends to perform the computation locally – may also necessitate changes in the planned path so that the drone can perform an intermediate

stop to a depot station in order to switch batteries. Last but not least, the time needed for a drone to cover a certain distance may not be known with full certainty when the mission is planned and the actual flight times may be different than those assumed when constructing the mission plan.

The main contributions of this work are as follows: (i) we capture the salient aspects of the problem at hand using a formal system model; (ii) we propose different heuristics to schedule such missions offline without considering any energy limitations; (iii) we evaluate those heuristics in missions where the type of drone and type of mission performed by each drone vary; (iv) we show that the mission time can be greatly optimized vs simpler policies for computational-resource sharing; (v) we propose a heuristic that tackles the problem considering the drones' autonomy; (vi) we evaluate the heuristic in a wide variety of experiments under different degrees of autonomy and battery changing delays; (vii) we describe an online protocol allowing autonomous drones to take online offloading decisions, in order to deal with any uncertainties in the flight times; (viii) the protocol is decentralized, and does not involve a global coordinator nor does it require any interaction between drones; (ix) we evaluate the protocol through extensive simulation experiments, using realistic system parameters; (x) we show that the proposed approach can significantly reduce the mission time of all drones in an even way.

The rest of the chapter is structured as follows. Section 3.2 gives an overview of related work. Section 3.3 provides the system model and provides a formal description of the problem. In Section 3.4, we propose a greedy heuristic with different offloading policies to tackle the problem assuming the drones have unlimited autonomy, while in Section 3.5 we describe EPPOS, an iterative algorithm that tackles the problem under energy limitations. Section 3.6 evaluates the greedy heuristic along with a simpler version of EPPOS for problems that the drones' autonomy suffice for the whole mission. Then, Section 3.7 evaluates EPPOS for scenarios where the drones' autonomy does not suffice for the whole mission, forcing the drones to make intermediate stops to change batteries. Finally, Section 3.8 presents the proposed online protocol for tackling the case where there is uncertainty for the drones' travel times, while Section 3.9 presents its evaluation.

3.2 Related Work

3.2.1 UAV as a server

A large group of work considers drones as edge servers, trying to serve users by optimizing the trajectory of the drones and the associated resource allocation. Such a problem is studied in [63] where a UAV is used as a base station for providing video content to multiple users. The authors of [64] investigate a problem where mobile devices that are not covered by an infrastructure may offload their computation to an edge-mounted UAV. The goal is to minimize the overall energy needed for IoT devices to complete their tasks by making decisions about the offloading, resource allocation and UAV trajectory. Both works split the problem in smaller sub-problems that are solved iteratively until convergence is reached in the optimization criteria. In [65], the authors address the problem where a UAV is used as an edge server for multiple mobile devices in order to offload their tasks. The goal is to jointly optimize the UAV trajectory and the computation offloading decisions and resource allocation to minimize the average weighted energy consumption of the UAV and the mobile devices. As above, this work subdivides the problem into smaller parts that are solved in an iterative way. Our work follows a similar approach to reduce complexity, by separating the path optimization from the resource allocation aspect, as each problem is hard on its own. Another work that considers a UAV as a server is [66], where the UAV is used to serve content to moving vehicles in a highway, by moving to and hovering over different locations, sending and receiving data to/from clients, and using a data cache with a limited capacity. The goal is to maximize the energy efficiency of the UAV by optimizing its movement, cache management and data transfers. In [67], multiple UAVs serve multiple IoT devices that need to offload their sensor data. The UAVs move to the proper locations in order to retrieve and process the data from the IoT devices. The goal is to serve all devices while minimizing the energy spent by the UAVs for flight and hovering. The main difference between all these problems and our work, is that drones play the role of a server whereas in our case drones are the clients of more powerful edge servers.

3.2.2 Resource allocation and offloading decisions

In [68], a drone must stop at predefined waypoints and perform heavyweight computations. The drone reduces its mission time by opportunistically offloading its computations to

an edge server when the offloading is beneficial. In [69], a drone-based system is used for tracking moving objects. The authors deal with the problem of deciding whether to offload a computation or execute it on the drone locally in order to minimize the response time. Unlike these works, which focus on runtime offloading decisions, we tackle the problem of planning the offloading decisions jointly with the drones' paths before the mission starts and adjusting it as needed at runtime.

The authors in [70] address the allocation of computing resources to services in a priority-aware way. They study two strategies, one that greedily considers services with higher priority, and one that assigns bigger weights to high-priority services while preventing the starvation of services with lower priority. In [71], a hybrid fog and cloud-aware heuristic is proposed, which attempts to offload computationally-heavy tasks to the cloud, and communication-heavy tasks to the fog. The offloading priority is set based on the task deadlines (earliest deadline first) and the VM to which the tasks are offloaded is the one with the earliest estimated completion time. A similar approach is adopted in [72] to orchestrate dynamically arriving task workflows in the fog. A workflow consists of tasks which need to perform a certain amount of computation followed by an amount of communication to send the data produced to another task down the line. We investigate a different optimization objective than the above works, proposing corresponding heuristics.

The work in [73] shows that simple algorithms, which are based on congestion estimation, can be used for task allocation in fog nodes and outperform the current nearest-node algorithm, leading to improved node usage and smaller job delays. We also estimate the expected contention on servers to plan resource allocation, but, in our case, the offloading reservations and respective allocations are planned offline, before the execution of the missions. [74] addresses the problem of orchestrating bag-of-tasks jobs in a set of processors. The tasks in a bag can run parallel to each other, and a task may dynamically spawn another task if its QoS is not sufficient. The authors test three allocation techniques, RR (round-robin), 2RC (2 random choices) and SQ (shortest queue), for allocating processors to tasks. The authors of [75] propose a scheme for deciding whether a computation will be offloaded to the edge or the cloud. Their scheme supports real-time constraints, and is configurable for better performance or lower cost. [76] focuses on minimizing the average service latency for multiple drones requesting service from edge servers, proposing an algorithm for making offloading decisions and resource allocation. Our heuristics also allocate the available server resources

to drones in a greedy way, but the decision is based on the expected completion time rather than the length of the server queues.

In [77], an algorithm for scheduling task graphs in a fog-cloud environment is proposed. The usage of the cloud comes with a financial cost. The goal is to schedule the tasks of a larger task graph so as to achieve a good trade-off between execution time and cost. The presented solution prioritizes the tasks based on their position in the task graph, then the best processing node is selected, and in the final phase tasks are reassigned to processing nodes to limit deadline violations. The authors of [78] focus on a problem where moving vehicles use edge servers to offload their tasks. They propose a decentralized method to find candidate servers in an area, and then propose a deadline and financial cost aware approach for choosing a server. In [79], a game-theoretical algorithm is proposed, where multiple users (players) that require offloading make choices about offloading to the edge or in a central cloud. The choices of the different users are coordinated in the cloud. The objective is to minimize the cost of using the edge and the cloud while respecting the delay constraints of the services offloaded. The authors of [80] propose an approach for scheduling dynamically arriving tasks with different priorities to edge servers. The goal is to minimize the average priority-weighted response time. Their algorithm consists of a decentralized approach for finding a proper server in combination with an online strategy for relocating tasks and reallocating resources. The relocation of tasks is done so that tasks with different priorities are executed to appropriate edge nodes.

A key difference of our online protocol with the above works, besides taking offloading decisions, is that we adapt the path of the drone at runtime because flight times critically affect the total mission times. Similar to [78] and [80], our approach works in a decentralized manner. We also use an opportunistic offloading approach for comparison with our proposed algorithm, which is similar to the shortest queue approach tested in [74]. Additionally, our approach tackles the incoming inquiries with priorities as in [80]. However, in our case, the servers do not communicate with each other, so a task relocation strategy cannot be adopted.

3.3 System Model & Problem Formulation

3.3.1 Drones, paths and flight model

Let there be M drones, $d_m, 1 \leq m \leq M$. Each drone is assigned an independent mission, in which it must visit a set of waypoints $\mathcal{V}_m$. These waypoints represent points of interest where the drone has to perform some sensing and then process the collected data on the spot, e.g., to take further action if a problem is detected.

To visit the assigned points of interest, each drone d_m follows a path P_m, which is encoded as a sequence of waypoints $P_m[i], 1 \leq i \leq \text{len}(P_m)$. The path must include all points of interest exactly once, $\exists! i : 1 \leq i \leq \text{len}(P_m) : P_m[i] = p, \forall p \in \mathcal{V}_m$. Also, each drone d_m initially starts its mission by taking off from a depot dep_m, and ends its mission by returning and landing at dep_m. We encode this in the drone's path by letting $P_m[1] = P_m[\text{len}(P_m)] = dep_m$. Without loss of generality, we assume the depot does not coincide with a point of interest, i.e., $dep_m \notin \mathcal{V}_m$. As will be discussed in the sequel, a drone may perform additional depot visits during its mission in order to switch batteries. Such intermediate depot visits, also referred to as depot detours, are explicitly encoded in the drone's path by inserting the depot as an intermediate waypoint.

Our flight model assumes polycopter drones, which have vertical take-off and landing capability, can fly between waypoints in a straight line, and can hover over a given position. Let $flyhT_m^{i,i+1}$ be the time needed for d_m to fly horizontally between $P_m[i]$ and $P_m[i+1]$. Further, let $takeoffT_m$ be the time needed for d_m to take-off from dep_m and reach the flight altitude for the mission at hand, and let $landT_m$ be the time needed to vertically land back at dep_m. These extra overheads are taken into account to derive the total flight delay between two successive waypoints in the drone's path, as follows:

$$flyT_m^{i,i+1} = \begin{cases} flyhT_m^{i,i+1}, & \text{if} P_m[i], P_m[i+1] \neq dep_m \\ flyhT_m^{i,i+1} + takeoffT_m, & \text{if} P_m[i] = dep_m \\ flyhT_m^{i,i+1} + landT_m, & \text{if} P_m[i+1] = dep_m \end{cases} \tag{3.1}$$

Note that the case where $P_m[i] = P_m[i+1] = dep_m$ is not handled above. This would mean that the drone takes-off from the depot only to immediately land back there, which cannot hold in a properly formed flight plan.

3.3.2 Sensing and computation

At each point of interest, the drone takes measurements via its onboard sensors, and then processes this data. We assume that each drone performs the same type of sensing and data processing at all points of interest. However, different drones may perform different types of sensing and data processing. Also, we assume that data processing needs to be performed in situ, while the drone is positioned over the point of interest, before moving on to the next waypoint of the mission.

Let $senseT_m$ be the time that is needed for d_m to perform the required sensing. Also, let $comp_m$ be the type of computation that must be performed on this data, taking $data_m^{in}$ amount of data as input and returning $data_m^{out}$ amount of data as a result. Notably, the type of computation determines the amount of resources res_m^{req} needed to run the computation on a computing platform.

Onboard computation

Each drone d_m has an onboard computer with hardware platform hw_m, which can be used to perform $comp_m$ locally. In order for the drone to be truly autonomous, its onboard platform has sufficient available resources to run the computation in question, $res_m^{avl} \geq res_m^{req}$. In this case, we assume that (local) data movement for $data_m^{in}$ and $data_m^{out}$ takes a negligible amount of time. Thus the total computation delay when processing is performed locally on the drone is $compT_m = procT(comp_m, hw_m)$.

Computation offloading

As another option, the drone can offload its computation to a nearby server located at the edge in order to accelerate processing and reduce the mission time. We assume that each edge server is suitably prepared to provide such a computation as a service. For instance, the respective software can be packaged and shipped to the servers in the form of micro-VMs or containers, before the drones start their missions.

Let there be K edge servers, $s_k, 1 \leq k \leq K$, at different locations in the wider area where the drones operate. Each server s_k may have a different hardware platform hw_k with available computing resources res_k^{avl}. Also, each server is accessible via a dedicated local wireless network with bandwidth bw_k and communication range $range_k$. When drone d_m visits a point of interest $P_m[i]$ which is in range of server s_k, it can offload its computation

$comp_m$ to it. Then, the computation delay for the drone is

$$compT(m,k) = procT(comp_m, hw_k) + \frac{data_m^{in} + data_m^{out}}{bw_k} \tag{3.2}$$

taking into account the time needed to perform the computation on the server and transfer the respective input and output over the network. For convenience, let $compT(m,0) = compT_m$ (the server identifier 0 means "no offloading").

Let $S_m[i]$ encode the offloading for d_m during its mission. More specifically, $S_m[i] = k$ if the drone shall offload $comp_m$ to s_k at $P_m[i]$, else $S_m[i] = 0$ if the drone shall perform the computation locally. Also, $S_m[i] = 0$ for $P_m[i] = dep_m$ as the drone does not perform any computation at the depot station.

Note that the drone may not be able to *immediately* offload its computation to $S_m[i]$ as soon as it completes sensing at $P_m[i]$. The reason is that server's resources may be used by other drones at that time. It is, however, possible for the drone to wait for the server to become available, and then proceed with the offloading process as usual. Let the extra waiting time for this be $W_m[i] \geq 0$. We let $W_m[i] = 0$ for all waypoints where $S_m[i] = 0$ and the drone does not use any server for offloading (the computation is performed locally on the drone without any extra waiting time).

Based on the above, the time spent by the drone in order to perform the required sensing and computation at each point of interest $P_m[i] \in \mathcal{V}_m$, is

$$pT_m^i = senseT_m + W_m[i] + compT(m, S_m[i]) \tag{3.3}$$

Server resource usage constraint

Let matrix U encode the planned usage of the server infrastructure, where $U[k][m][t] = 1$ if s_k allocates resources to run the service for $comp_m$ at time t, else $U[k][m][t] = 0$. Note that the total amount of resources allocated on a server due to offloading cannot exceed its overall resource capacity:

$$res_k^{avl} \geq \sum_{m=1}^{M} U[k][m][t] \times res_m^{req}, \forall k, t \tag{3.4}$$

We assume that the resources res_m^{req} associated with the service for performing $comp_m$, remain allocated only while the service is actually being used by a drone. More specifically,

if d_m starts offloading its computation to s_k at time t_{start}, it will consume resources during the entire duration of the computation including the required input/output data transfers, $U[k][m][t] = 1, t_{start} \leq t \leq t_{start} + compT(m,k)$. Notably, in this work, we assume that an edge server cannot further offload a computation to another edge server or the cloud.

3.3.3 Energy model

We assume battery-operated drones (however, our model can be equally applied to drones with an engine and a fuel tank). Let E_m^{max} be the maximum energy storage capacity of d_m's battery. Obviously, bigger values of E_m^{max} translate to larger autonomy, allowing the drone to operate for a longer amount of time before it needs to land.

Let the energy that is consumed by d_m to fly between two waypoints $P[i]$ and $P[i+1]$ be a linear function of flight time, $flyE_m^{i,i+1} = \alpha_m \times flyT_m^{i,i+1}$. In the same spirit, the energy consumed by the drone to hover above $P[i]$ is a linear function of the time spent at that point in order to perform the required sensing and computing task, $pE_m^i = \beta_m \times pT_m^i$. We consider the energy spent for onboard computation or the communication with a server to be negligible compared to the energy spent for hovering. Also, for polycopter drones flying at moderate speeds, we assume that $\alpha_m \approx \beta_m$, i.e., the energy spent for flying is comparable to the energy spent for hovering.

Safety constraint

Let $remE_m^i$ denote the remaining energy of d_m at $P_m[i]$, after sensing and processing. Initially, $remE_m^1 = E_m^{max}$ as every drone starts from its depot with full batteries. The remaining energy at the next point in the drone's path is equal to the energy that was available at the previous point less the energy spent to fly to the next waypoint and the energy spent there (to hover while performing the required sensing and processing):

$$remE_m^{i+1} = remE_m^i - flyE_m^{i,i+1} - pE_m^{i+1} \tag{3.5}$$

Note that it is crucial to build flight paths and make offloading decisions so that the below equation holds:

$$remE_m^i > 0, \forall i, m \tag{3.6}$$

The above constraint states that at no point during the execution of the mission should a drone deplete its batteries. Such an event would lead to an emergency landing, which is not only undesirable from a purely operational perspective but may also lead to material damages or even human injuries, depending on the area of operation.

Depot detours

To satisfy the above constraint, drones with limited autonomy or long missions may have to perform intermediate depot visits (detours) in order to switch batteries so that they can proceed with the rest of their mission. Let $depT_m$ denote the time it takes for d_m to switch batteries once it has landed at its depot dep_m. Taking this into account, we extend Equation 3.3 to redefine the time spent at each point in the drone's path as

$$pT_m^i = \begin{cases} senseT_m + W_m[i] + compT(m, S_m[i]), & \text{if } P_m[i] \neq dep_m \\ depT_m, & \text{if } P_m[i] = dep_m \wedge 1 < i < \text{len}(P_m) \\ 0, & \text{otherwise} \end{cases} \tag{3.7}$$

The first case captures the time spent at points of interest (all waypoints that do not represent a depot visit), as per Equation 3.3. The second case captures the time spent at a depot to switch the drone's batteries before it can continue its mission. If, however, the drone is at its depot in the beginning and end of its path, the third case applies, setting the time spent at the depot to 0. This is because each drone starts its mission with fully charged batteries and the mission is considered to be completed the moment the drone lands at its depot for the last time (the battery switch time after the mission's completion does count in the actual mission time).

In the same vein, we re-define the energy spent at each point in the path as

$$pE_m^i = \begin{cases} \beta_m \times pT_m^i, & \text{if } P_m[i] \neq dep_m \\ -E_m^{max} + remE_m^{i-1} - flyE_m^{i-1,i}, & \text{if } P_m[i] = dep_m \wedge remE_m^{i-1} - flyE_m^{i-1,i} > 0 \\ 0, & \text{otherwise} \end{cases} \tag{3.8}$$

The first case applies to all points of interest, as already discussed above. The battery switching at the depot is captured by the second case, which sets the energy spent to a *negative* value so that the remaining energy of the drone $remE_m^i$ after a successful depot visit is equal

to E_m^{max} as per Equation 3.5. Note, however, that this artificial correction is allowed only if the drone has sufficient energy to reach the depot in the first place, as ensured by the second term of the condition for this case. Else, the third case of the above equation applies, which sets the energy spent at the depot to 0 so that $remE_m^i \leq 0$ as per Equation 3.5, indicating that the drone will deplete its batteries before reaching the depot. As already discussed, this should never be the case in a properly planned mission.

3.3.4 Problem

Given the above, the time needed for d_m to complete its entire mission can be captured as:

$$mT_m = \sum_{i=1}^{\text{len}(P_m)-1} flyT_m^{i,i+1} + pT_m^{i+1} \tag{3.9}$$

Our goal is to exploit the available edge server infrastructure in order to reduce the mission time of all drones. More specifically, the problem we tackle in this work is the following:

Given drones $d_m, 1 \leq m \leq M$ with assigned points of interest $\mathcal{V}_m$ where they have to perform some sensing and data processing, and stationary servers $s_k, 1 \leq k \leq K$ located in the mission area, which can be used for offloading the drones' computations, produce paths P_m and offloading plans S_m that minimize mT_m (as per Equation 3.9) *in a fair way for all drones*, while ensuring that the constraints of Equation 3.4 and Equation 3.6 are satisfied.

We capture the fairness objective by using as a reference the makespan of a default path P_m^{def} that does not involve any computation offloading. The corresponding mission time mT_m^{def} can be computed as per Equation 3.9 for a special "empty" offloading schedule S^{def} where $S_m^{def}[i] = 0, 1 \leq i \leq \text{len}(P_m^{def})$. Based on this reference, the relative reduction of the default mission time that is achieved by an alternative plan is:

$$red_m = \frac{mT_m^{def} - mT_m}{mT_m^{def}} \tag{3.10}$$

Based on this, we set as the optimization target:

$$\text{maximize } min_{m=1}^{M}(red_m) \tag{3.11}$$

In other words, we wish to maximize, across all drones, the relative reduction of their mission time, using the available server infrastructure for offloading. The rationale is for all concur-

rently running missions to benefit from the shared server resources in an even way.

3.3.5 Uncertain flight times variant

In this work, we also tackle the scenario where the flight times between two waypoints may not be known with certainty during the planning phase. Instead, we assume that the actual flight times vary according to a uniform random distribution with upper bound $flyMaxT_m^{i,i+i}$ and lower bound $flyMinT_m^{i,i+i}$. Note that this randomness directly affects the corresponding energy consumption. To deal with this uncertainty, we use as a starting point the mission plan produced by the offline algorithm for the worst-possible flight times and energy consumption, and propose an ad-hoc protocol for deciding, at runtime, whether and to which edge server each drone shall offload its computation at each point of interest $P[i] \in \mathcal{V}_m$. As a result, the drone may pick a different offloading option than the one suggested by the offline plan. In addition, any time/energy savings achieved vs the offline plan are exploited to postpone or even eliminate depot detours so as to further reduce the mission times.

We assume that the drone's maximum battery capacity E_m^{max} is sufficient so that d_m can move from its depot dep_m (starting with full batteries) to any point of interest $P_m[i]$ and back to dep_m even if the flight times are the highest possible and the computation at that point is performed locally on the drone. This is to ensure that the problem always has a feasible solution irrespective of the points of interest assigned to each drone and the degree of contention among drones for the shared server resources.

3.4 Greedy Approach

In this section, we deal with the simpler version of the problem without considering any energy limitations or uncertainties. We describe the proposed heuristic that works in a greedy manner.

The algorithm takes as input a default path P_m^{def} for each drone d_m, generated separately for each drone using a conventional TSP algorithm, which includes the required visitations at all points of interest $\mathcal{V}_m$, without any computation offloading. Note that we tackle applications where the drone does not have to follow a pre-specified path. It is merely required to visit all specified points of interest in any order. Thus, P^{def} is merely indicative.

As a first step, the algorithm manipulates the default paths P^{def} so as to reduce the overall

contention of the drones for the edge resources. This is done via function ReversePaths(), which works in an iterative way, by randomly picking a drone and exploring whether a complete path reversal (visiting the points of interest in reverse order) would reduce the contention of the system. If so, the reversed path is adopted instead of the default (note that such a path reversal does not change the default mission time). The expected contention is roughly estimated based on the time periods where more than one drone need to perform computations at points of interest in range of the same servers, without considering any actual offloading decisions that might be taken at a later stage.

The algorithm then incrementally introduces offloading in an iterative way. In each iteration, a drone is picked and the next visit is planned for it, updating the drone's state and server resource usage accordingly. If the end of the path is reached for a drone, it is removed from the set of candidates considered for the next iteration. Finally, the algorithm terminates when there are no candidates left. The algorithm's core logic is given in Algorithm 7.

Function PickDrone() selects the drone to be considered in the current iteration of the algorithm (we experiment with three different policies, discussed in more detail in the sequel). Function PlanNxtVisit() plans the visit to the next point in the drone's path and updates its mission time. The possibility to offload the drone's computation at that point is examined in ReserveOffload(), which greedily finds via function nxtOffload() the best available server and associated waiting time, which minimizes Equation 3.3. If found, the resources needed for the computation are reserved on the server, and the drone's offload plan is updated accordingly.

We experiment with three policies for selecting the next drone to be considered in the top-level planning loop, implemented as different versions of function PickDrone(): (i) round-robin (RR), (ii) current benefit (CB), and (iii) expected benefit (EB). The RR policy simply picks the next drone in the candidate list (where drones are initially placed in random order) in a blind round-robin fashion. The CB policy picks the drone that has currently benefited the least from the available server infrastructure in terms of the relative reduction of its default mission time thanks to offloading. More specifically, CB picks the d_m with the minimum $\frac{mT_m^{def} - mT_m^{upd}}{mT_m^{def}}$, where mT_m^{upd} is initially set equal to mT_m^{def} and is updated after a computation offloading is planned in the ReserveOffload() function.

Finally, the intuition behind the EB policy is the following. Assume there are two drones, d_1 and d_2, with $len(P_1^{def}) < len(P_2^{def})$. Also, assume that a single server is accessible from

Algorithm 7 Greedy algorithm.

function MAIN($\mathcal{D}, P^{def}$)
  $P \leftarrow$ REVERSEPATHS(P^{def})
  for each d_m **do**
    $curr_m, t_m \leftarrow 1, 0$ ▷ start from 1st waypoint, at time zero
    $S_m[i] = 0, 1 \leq i \leq \text{len}(P_m)$ ▷ no offloading, yet
    $W_m[i] = 0, 1 \leq i \leq \text{len}(P_m)$ ▷ no waiting times, yet
    $U[k][m][t] \leftarrow 0, \forall k, t$ ▷ no reservations, yet
  end for
  while $\mathcal{D} \neq \emptyset$ **do**
    $d_m \leftarrow$ PICKDRONE($\mathcal{D}$)
    PLANNXTVISIT(d_m, U)
    if $curr_m = \text{len}(P_m)$ **then** ▷ end of mission
      $\mathcal{D} \leftarrow \mathcal{D} - d_m$
    end if
  end while
end function

function REVERSEPATHS(M, P^{def})
  $P \leftarrow P^{def}$
  for M **do**
    $m \leftarrow$ pickRandom(M)
    $P' \leftarrow P - P_m +$ reversePath(P_m)
    if calcContention(P') $<$ calcContention(P) **then**
      $P \leftarrow P'$
    end if
  end for
  return P
end function

function PLANNXTVISIT(d_m, U)
  $i \leftarrow curr_m$
  $t_m \leftarrow t_m + flyT_m^{i,i+1} + senseT_m$
  $k \leftarrow$ RESERVEOFFLOAD($d_m, t, i+1, U$)
  $t_m \leftarrow t_m + pT_m^{i+1}$
  $curr_m \leftarrow i + 1$
end function

function RESERVEOFFLOAD(d_m, t, i, U)
  $(k, t_{start}) \leftarrow$ nxtOffload($P_m[i], t, comp_m, U$)
  if $k \neq 0 \wedge t_{start} - t + compT(m, k) < compT(m, 0)$ **then**
    $U[k][m][t] \leftarrow 1, t_{start} \leq t \leq t_{start} + compT(m, k)$
    $S_m[i] \leftarrow k$
    $W_m[i] \leftarrow t_{start} - t$
  end if
  return k
end function

all points of interest, and that if one of the two drones offloads its computation to the server then the other drone loses this offloading opportunity. Consequently, d_1 competes with d_2 for

server usage during its entire mission, whereas d_2's mission includes a contention-free part where the drone can freely offload its computations to the server. Note that in this case RR will evenly allocate the server to d_1 and d_2 during the period of contention, while CB will even tend to favor d_2 over d_1. Obviously, both approaches are not optimal with respect to the objective expressed in Equation 3.11. A better approach would be to give (some) preference to d_1 during the period of contention since d_2 will be able to freely use the server for the remainder of its mission without any contention. To this end, one needs to consider the offloading opportunities and level of contention each drone will have during its mission.

More specifically, in each iteration, EB picks the drone d_m with the minimum $\frac{mT_m^{def} - mT_m^{exp}}{mT_m^{def}}$, where mT_m^{exp} is the expected mission time. This is estimated based on the computation offloading that has been planned so far, and the expected offloading decisions that will be taken in the future as part of the planning algorithm, for each point in the drone's path.

To determine the expected offloading decision at point $P_m[i]$ where d_m is expected to arrive at time t, first we estimate the potential contention $cont_{m,i}$ at this point, as the number of drones that are expected to arrive at (perhaps other) points of interest covered by the same servers at a time that is "close" to t (where offloading is still beneficial despite the potential waiting time due to server overload), divided by the number of servers that cover these points. Then, the probability for d_m to be able to offload its computation at $P_m[i]$ is

$$P_{m,i}^{offld} = \begin{cases} 1 & \text{if } cont_{m,i} \leq \Upsilon_i \\ \frac{w_{m,i} \times \Upsilon_i}{cont_{m,i}} & \text{otherwise} \end{cases} \tag{3.12}$$

with $\Upsilon_i = \frac{avgFlyT}{avgCompT_i} \times avgCapacity_i$, where $avgFlyT$ is the average flight time between waypoints (over all drones), and $avgCompT_i$ is the average time needed by the servers reachable from $P_m[i]$ to perform the computations of the drones that can use these servers for offloading at the same time as d_m, and $avgCapacity_i$ is the average capacity of these servers. The factor $\frac{avgFlyT}{avgCompT_i}$ stands for the number of computations that can be performed by the servers sequentially before any drones that are in transit might require the server for offloading (thereby leading to a potential contention situation). Also, $w_{m,i} = \frac{len(P_m)-i}{len(P_m)}$ is a weight that is higher for the points of interest early in the drone's path and decreases for the ones further down the line. The rationale is for the offload estimation to be more pessimistic for the far future (if there is contention) as there is increased uncertainty about the decisions that will be taken by the algorithm at distant points.

Finally, the expected total mission time is estimated by using, at each point $P_m[i]$, Equation 3.12 to compute the probability $P_{m,i}^{offld}$ for d_m to offload the computation to a server, or to perform the computation locally with probability $1 - P_{m,i}^{offld}$.

3.5 Energy-aware Path Planning and Offloading

Next we tackle the problem, considering the energy limitations of the drones. To do this with acceptable complexity, we propose an energy-aware path planning and offload scheduling heuristic (EPPOS). The heuristic works in the spirit of a variable neighborhood search (VNS) algorithm [81], gradually improving the best solution found so far by exploring neighboring solutions through random search. In our case, the solution consists of (a) the paths to be followed by the drones, and (b) the schedule for offloading their computations to nearby edge servers. In turn, the offloading schedule is produced based on the expected computation time at each point of interest in the drone's path, and the so-called candidate selection order in which the drones are examined to find the next best possible offloading option. The latter is encoded as a sequence of drone identifiers, e.g. $[1, 2, 3.., 1, 2, 3..]$ means that the algorithm will first plan the next offloading option for drone d_1, then for d_2, then for d_3 etc. Note that the same identifier may appear more than once in this sequence, each such occurrence dictating when the algorithm will plan the next offload for that drone with respect to others.

3.5.1 Description

The high-level logic of the heuristic is given in Algorithm 8 while the main auxiliary functions are described in Algorithm 9. The algorithm takes as input the set of drones $\mathcal{D}$, an initial path P_m^{init} for each drone d_m (computed using an off-the-shelf TSP algorithm without considering energy constraints or computation offloading options), and the number of iterations of the optimization loop.

As a first step, it computes for each drone d_m, the expected time spent at each point of interest, let $pT_m^{exp}[i]$. This is set equal to the sensing time $senseT_m$ plus the average computation time over all servers in range of $P_m[i]$ and the local computation at d_m (assuming that each option is equally probable and that d_m will not have to wait before starting the computation for that point of interest). Further, the candidate selection order is set to a randomly chosen round-robin order.

Algorithm 8 Energy-aware path planning and offload scheduling (EPPOS)

function MAIN($\mathcal{D}, P^{init}, iterations$)
  for each $d_m \in \mathcal{D}$ **do**
    for each $P_m[i] \neq dep_m$ **do**
      $pT_m^{exp}[i] \leftarrow senseT_m +$ avg$(compT(m,k)), \forall s_k$ in range of $P_m[i]$ and $k = 0$
    end for
  end for
  $order \leftarrow rndRoundRobinOrder(\mathcal{D})$
  $bestMinReduction \leftarrow -1$

  for $iterations$ **do**
    $P \leftarrow$ ADJUSTPATHS($\mathcal{D}, P^{init}, pT^{exp}$)
    $P, S, W, U, reductions \leftarrow$ SCHEDULEOFFLOADING$(\mathcal{D}, P, pT^{exp}, order)$
    $minReduction \leftarrow$ min$(reductions)$
    if $minReduction > bestMinReduction$ **then**
      $bestMinReduction \leftarrow minReduction$
      $bestP, bestS, bestW, bestU \leftarrow P, S, W, U$
      $bestOrder, bestpT^{exp} \leftarrow order, pT^{exp}$
    end if
    $order \leftarrow$ SHAKEORDER$(bestOrder, reductions)$
    $pT^{exp} \leftarrow$ ADAPTEXPTIMES$(bestpT^{exp})$
  end for
end function

function ADJUSTPATHS($\mathcal{D}, P, pT^{exp}$)
  for each $d_m \in \mathcal{D}$ **do**
    if isUnsafe(d_m, P_m, pT_m^{exp}) **then**
      insertDepotDetours(P_m)
      trySwapReverseSubPaths(P_m) ▷ if this reduces contention
    else
      tryReversePath(P_m) ▷ if this reduces contention
    end if
  end for
  return P
end function

Then the algorithm iteratively improves the paths and offloading schedule. Each iteration starts by resetting the path P_m of each d_m to P_m^{init} and adjusting these paths based on the expected times pT_m^{exp} spent at each point of interest, via function ADJUSTPATHS(). More specifically, if the expected remaining energy of d_m does not suffice to reach all points of interest, its path P_m is considered unsafe and one or more intermediate depot visits (detours) are inserted at the proper points in order to switch batteries. This is done by finding the next waypoint in the path that cannot be reached because the drone will have exhausted its battery, and then checking the previous waypoints in the path (up to the last depot visit) to insert a

Algorithm 9 Auxiliary functions

function SCHEDULEOFFLOADING($\mathcal{D}, P, pT^{exp}, order$)
  for each $d_m \in \mathcal{D}$ **do**
    $pos_m, remE_m^1 \leftarrow 1, E_m^{max}$ ▷ start from $P_m[1] = dep_m$ with full batteries
    $S_m[i] = 0, W_m[i] = 0, 1 \leq i \leq \text{len}(P_m)$ ▷ no offloading / waiting times yet
    $U[k][m][t] \leftarrow 0, \forall k, t$ ▷ no server reservations yet
  end for
  while $order \neq \emptyset$ **do**
    $d_m \leftarrow$ pickNxtDrone($order$)
    $pos_m, P_m, k, t^{arr}, waitT \leftarrow$ fndNxtOffload($pos_m, P_m, pT^{exp}, S, W, U$)
    if $pos_m \neq len(P_m)$ **then** ▷ offload option found
      $S_m[pos_m], W_m[pos_m] \leftarrow k, waitT$
      $U[k][m][t] \leftarrow 1, \forall t, t^{arr} + waitT \leq t \leq t^{arr} + waitT + compT(m, k)$
    else ▷ arrived at the end of the path
      $order \leftarrow$ rmvAllEntries($order, d_m$)
      $reductions_m \leftarrow \frac{mT_m^{def} - mT_m}{mT_m^{def}}$ ▷ reduction as per Equations3.9 and 3.10
    end if
  end while
  return $P, S, W, U, reductions$
end function

function SHAKEORDER($order, reductions$)
  $moves \leftarrow$ pickRndRange($moves^{low}, moves^{high}$)
  $bestD \leftarrow$ sortDiscendingReductions($reductions, \mathcal{D}$)
  $worstD \leftarrow$ sortAscendingReductions($reductions, \mathcal{D}$)
  for m **from** 1 **to** $moves$ **do**
    $first\_occ \leftarrow$ pickRndRange($first\_occ^{low}, first\_occ^{high}$)
    $order \leftarrow$ moveFirstEntriesToTail($order, bestD[m], first\_occ$)
    $last\_occ \leftarrow$ pickRndRange($last\_occ^{low}, last\_occ^{high}$)
    $order \leftarrow$ moveLastEntriesToHead($order, worstD[m], last\_occ$)
  end for
  if pickRndBinary($prob_{swap}$) $= true$ **then**
    $swaps \leftarrow$ pickRndRange($swaps^{low}, swaps^{high}$)
    for $swaps$ **do**
      $m_1, m_2 \leftarrow$ pickRndPair($order$)
      $order \leftarrow$ swapEntries($order, m_1, m_2$)
    end for
  end if
  return $order$
end function

function ADAPTEXPTIMES($bestpT^{exp}$)
  if pickRndBinary($prob_{pTchng}$) $= true$ **then**
    $pTchng \leftarrow pickRndRange[pTchng^{low}, pTchng^{high}]$
    $pT_m^{exp}[i] \leftarrow bestpT_m^{exp}[i] \times pTchng$ ▷ $\forall P_m[i] \neq dep_m$ in range of a server
  else
    $pT_m^{exp}[i] \leftarrow bestpT_m^{exp}[i]$ ▷ $\forall P_m[i] \neq dep_m$ in range of a server
  end if
  return pT^{exp}
end function

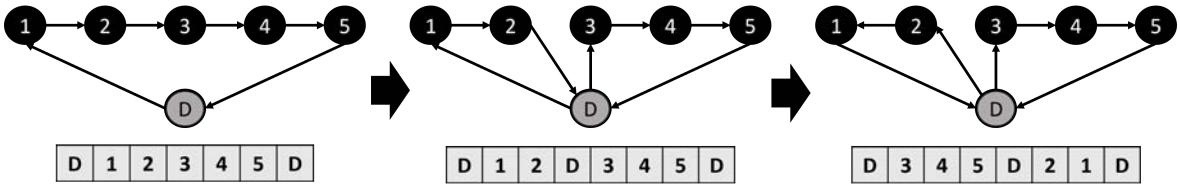


Figure 3.1: Path adjustment example. The initial main path $[p_1, p_2, p_3, p_4, p_5]$ (left) is considered unsafe and as a consequence a depot detour is inserted between point of interest p_2 and p_3, resulting in two subpaths $[p_1, p_2]$ and $[p_3, p_4, p_5]$ (center). In addition, to minimize contention, the two resulting subpaths are swapped and the subpath $[p_1, p_2]$ is reversed and becomes $[p_2, p_1]$ (right).

detour between two waypoints so as to minimize the extra flight delay. As a result of such detours, P_m is effectively split in smaller subpaths separated by depot visits. In this case, the algorithm considers swapping the order of these subpaths as well as reversing the order in which the different points of interest are visited within each subpath, to reduce the expected contention for the available server resources. More precisely, the contention at each waypoint $P_m[i]$ is estimated by dividing the number of drones expected to arrive at points of interest covered by the servers that are in range of $P_m[i]$ (where the arrival time falls within the time window where offloading is still beneficial despite the potential waiting time due to server overload), with the number of servers that cover $P_m[i]$. High contention at a point of interest means that the drone has lower probability to successfully offload its computation to some server. To avoid performing an exhaustive search for this, we only consider a limited (up to a fixed maximum) number of swap/reversal samples, which are randomly selected in each iteration. Else, if P_m is safe (no depot detour is added), the algorithm simply considers a reversal of the entire path in order to reduce contention. These path adjustments are illustrated in Figure 3.1. Note that these adjustments are applied to the P_m only if they reduce the average contention along that path.

After the paths are reset and adjusted, a new offloading schedule is produced via the ScheduleOffloading() function. This starts from a fresh state where all servers are unoccupied at all times and no computation offloading is planned for any drone. Then, drones are considered according to the candidate selection order, and for each one the next computation offload is planned. This is done via the fndNxtOffload() function, which finds the next best safe offloading option in the drone's path (not shown for brevity). Notably, this function transparently handles the case where, as a side-effect of offloading decisions taken in previous steps, the actual time spent by d_m at some point of interest $P_m[i]$ may turn out to

Parameter	Description
$moves^{low/high}$	Bounds for the random pick of the number of best and worst candidates to be moved to the tail and head of the candidate order/selection list, respectively.
$first\_occ^{low/high}$	Bounds for the random pick of the number of occurrences of the best candidate to be moved to the tail of the order list.
$last\_occ^{low/high}$	Bounds for the random pick of the number of occurrences of the worst candidate to be moved to the head of the order list.
$prob_{swap}$	Probability to swap entries in the order list.
$swaps^{low/high}$	Bounds for the random pick of the number of swaps to be performed in the order list.
$prob_{pTchng}$	Probability for changing $bestpT^{exp}$ to be used as input for the next iteration
$pTchng^{low/high}$	Bounds for the random pick of the change factor for $bestpT^{exp}$.

Table 3.1: Parameters guiding the random search of the EPPOS.

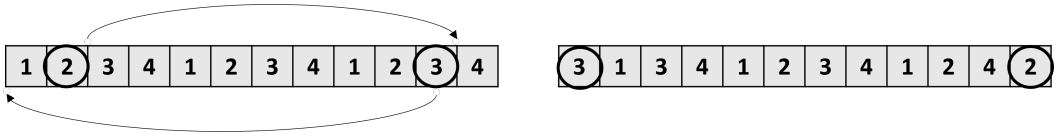
be different than what was expected, $pT_m[i] \neq pT_m^{exp}[i]$. If such individual or accumulated deviations render P_m infeasible, an intermediate depot detour is inserted in P_m in the drone's path to ensure the safety constraint as per Equation 3.6 and the search for the next best offloading option is repeated. In any case, the function returns a possibly updated path and the offloading suggestion at the next possible point of interest, along with the respective server and waiting time so that the server capacity constraint as per Equation 3.4 is satisfied. If the returned point indeed corresponds to a point of interest, the offloading schedule is updated accordingly. Else, the returned point corresponds to the last point in the drone's path implying that no offloading opportunity was found for d_m, in which case all occurrences of the drone are removed from the ordered candidate selection list.

When the ScheduleOffloading() function returns, it is checked whether the newly produced paths and offloading schedule improves the optimization criterion vs the best solution found so far (achieves a higher worst relative reduction of the mission time across all drones). If so, the new paths and schedule, along with the corresponding candidate selection order and expected time spent by each drone at each point of interest, are stored as the best solution.

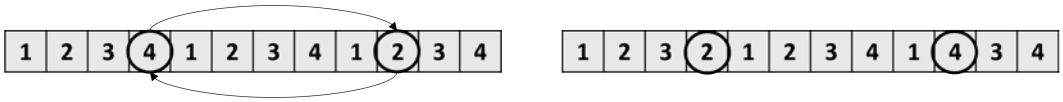
Finally, before entering the next iteration, the best candidate selection order and best expected times at each point are modified via the ShakeOrder() and AdaptExpTimes() function, respectively. These functions randomly change these crucial parameters in search for a better solution, and are briefly discussed in the following. Table 3.1 lists the parameters that guide this random search.

Function ShakeOrder() changes the best selection order found so far to build a new

selection order in which the drones will be considered for offloading in the next iteration. This is done in two different ways. On the one hand, it picks $moves$ drones that achieved the highest relative reduction of their mission times and for each one moves its $first\_occ$ first occurrences in the order list to the tail of the list. Also, it picks the same number of drones that achieved the lowest relative reduction of their mission times and for each one moves its $last\_occ$ last occurrences in the order list to the head of the list. On the other hand, with probability $prob_{swap}$, it swaps the order of two randomly chosen list entries. This is repeated for $swaps$ times. Figure 3.2 illustrates simplified scenarios for those operations. The rationale of the first transformation is to strike a better balance by promoting the drones with the worst mission time improvement so that they are considered earlier for offloading in the next iteration, while demoting the drones with the best improvement. The second transformation is a simple random mutation.



(a) Move the first occurrence of the drone with the highest relative reduction of the mission time, let d_2, to the tail of the list, and move the last occurrence of the drone with the lowest relative reduction, let d_3, to the head of the list.



(b) Throw a binary dice which gives $true$ with probability $prob_{swap}$. If $true$, pick two random occurrences in the list, e.g., the first occurrence of d_4 and the last occurrence of d_2, and swap them.

Figure 3.2: Shake operations (mutations) on the order / candidate selection list (for $moves = 1$, $first\_occ = 1$, $last\_occ = 1$, $swaps = 1$).

Function AdaptExpTimes() adapts the expected time that will be spent by the drones at each point of interest, based on the values that were used to find the best solution so far. More specifically, with probability $prob_{pTchng}$, the expected times at each point of interest for the next iterations are set to $bestpT_m^{exp}[i]$ adapted by a factor $pTchng$, else they are set equal to $bestpT_m^{exp}[i]$. The underlying rationale for this adaptation is to compensate for the fact that these estimates are based on rough approximations for the time that will be spent at each point of interest to perform the computation. Note that all the above variables ($moves$, $first\_occ$, $last\_occ$, $swaps$, $pTchng$) are randomly picked from respective intervals that are specified via the bounds given in Table 3.1.

3.5.2 Complexity

The runtime complexity of the EPPOS heuristic is determined by the number of iterations and the complexity of the procedures that are executed in each iteration, briefly discussed below. For brevity we let $V = \sum_{m=1}^{M} |\mathcal{V}_m|$ denote the total number of visits to points of interest to be performed by the drones. Also, let $D = \frac{\sum_{m=1}^{M} max_{s_k \text{ in range of } \mathcal{V}_m}(compT_m - compT(m,k))}{M}$ be the maximum difference between the local computation time of a drone and the computation time at the fastest server in range of a point of interest, averaged over all drones. Note that D corresponds to the average number of checks that are needed to determine whether a drone can offload its computation to a server at an acceptable waiting time so that this is still beneficial compared to performing the computation locally.

The complexity of the path adjustment procedure (AdjustPaths function) is $O(V) + O(M \times V \times D \times K)$, for the insertion of depot detours and the consideration of path reversal/swaps, respectively. In the former case, the path of each drone needs to be traversed linearly (with limited backwards checks that do not increase the complexity). In the latter case, for each drone, a fixed number of subpath reversal/swap combinations are checked for contention against the path of every other drone. In turn, the contention check at each point of interest is done for all servers in range and for the time slots where computation offloading remains beneficial. Resource allocation (ScheduleOffloading function) has a complexity of $O(V \times D \times K)$ because for each point of interest in the path of each drone, all servers in range are checked for availability for the time slots where offloading is still beneficial, in order to decide whether, where and when to offload the computation of the drone. The random adaptation of the expected time spent by each drone at each point of interest (AdaptExpTimes function) has a complexity of $O(V)$. Finally, the complexity of the random rearrangement of the candidate selection list (ShakeOrder function) is $O(V)$ as the total number of performed mutations (regulated through the random variables $moves$, $first_{occ}$, $last_{occ}$ and $swaps$) cannot be larger than V.

It follows that the aggregate complexity of an iteration is $O(M \times V \times D \times K)$. Even though this is not negligible, it is relatively lightweight compared to an exhaustive check of all possible drone path and offloading combinations, allowing the algorithm to scale for larger instances of the problem.

3.6 Evaluation of the Heuristics - no Energy Constraints

We evaluate the proposed heuristics via simulations, based on performance numbers that are obtained in a realistic way. We present our evaluation based on two main sets of experiments. First we evaluate the greedy heuristic when the drones operating in the target area are homogeneous. Then we continue our experimentation assuming heterogeneous drones in terms of flying and processing capabilities.

3.6.1 Topology and missions

We conduct experiments for a 21×21 grid topology, shown in Figure 3.3. The nodes represent the potential points of interest (waypoints) for the drones. The horizontal and vertical distance between neighboring nodes is 20 meters. All drones start from and land on the same depot node, located in the center of the grid. Further, we assume two edge servers, located at the gray black-bordered nodes, featuring WiFi interfaces with a range of 200 meters at a practically constant throughput of 52 Mbps (this is realistic according to network simulations performed via the NS3 [82] for TCP/IP communication over high gain antennas and line-of-sight without any obstacles or interference). The circular regions in Figure3.3 mark the regions from where each server is reachable and can be used for offloading.

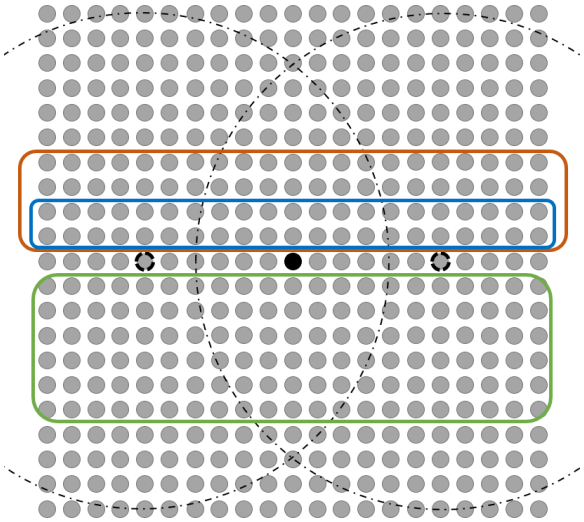


Figure 3.3: Topology used in the experiments.

We experiment with three types of missions, for different regions denoted in the figure via dashed rounded rectangles: the *small* mission which includes all points in the two rows above the central row of the grid (42 points of interest); the *medium* mission which includes

the four rows above the central row (84 points of interest); the *large* mission which includes the six rows bellow the central row (126 points of interest). The paths to be followed by the drones are computed using a conventional TSP algorithm and are subsequently fed into our offload planning heuristic.

3.6.2 Drone and server model parameters

We assume a fleet of drones that have the same hardware and perform the same sensing and processing task at each point of interest: take a picture using the onboard camera and process the image to detect certain objects. The sensing (image capture) delay $senseT$ is 1 second and the image obtained is $\sim$ 1MB. Object detection is done using YOLO [83]. For the local computation, as a typical hardware platform, we use an RPi 4 model B with 2GB of RAM running a headless Ubuntu server 20.04. The measured delay $compT_m$ is 10 seconds.

For the edge server we use a laptop with an i7-8550U CPU, running a VM with 4 cores and 4GB of RAM. The object detection service used for offloading runs inside the VM as a container using Docker [84]. The total computation delay $compT(m, k)$, including container loading and the time needed to transmit the request with the image and the detection results over WiFi, is 2 seconds. The number of drones that can be served concurrently by each such server is set to 1; this is because we are interested in examining scenarios with resource pressure on the edge servers, which may also be caused by external load; also we observed a significant increase in the processing time when two or more services run parallel in the VM.

The flight model used in our experiments assumes constant acceleration 0.4 m/s^2 when the drone departs from a waypoint until it reaches a speed of 4 m/s, and a constant deceleration 0.8 m/s^2 as it approaches and stops at the next waypoint. The resulting flight behavior is similar to what we have observed via the ArduPilot software-in-the-loop configuration [85].

With this model, the flying time between two neighboring waypoints is about 12 sec. Also, the default makespan mT_m^{def} (when the drone performs the required computation at all points of interest using its onboard platform, without any offloading) for the small, medium and large mission is roughly 17 min, 33 min and 49 min, respectively.

3.6.3 Homogeneous drones

In the first set of experiments the drones operating in the target area, are homogeneous in terms of flying speed and processing capabilities. Here we evaluate the performance of the

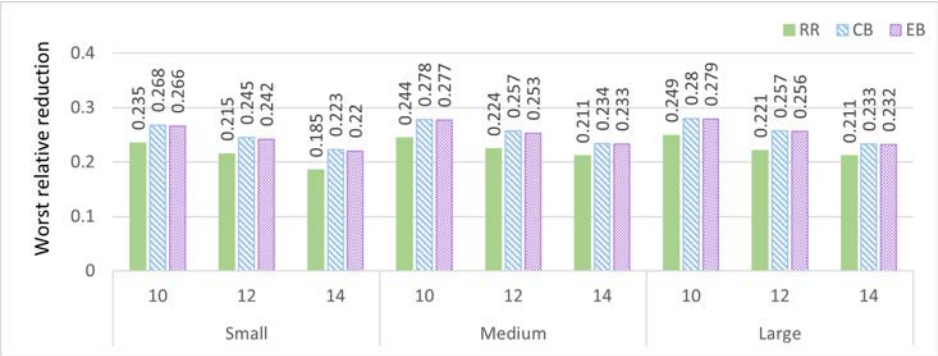
greedy heuristic. We start by examining the case where the paths remain as specified in order to evaluate the performance of the RR, CB and EB policies. Then, we present and discuss the results for the path optimization discussed in the Section 3.4.

Evaluation of selection policies

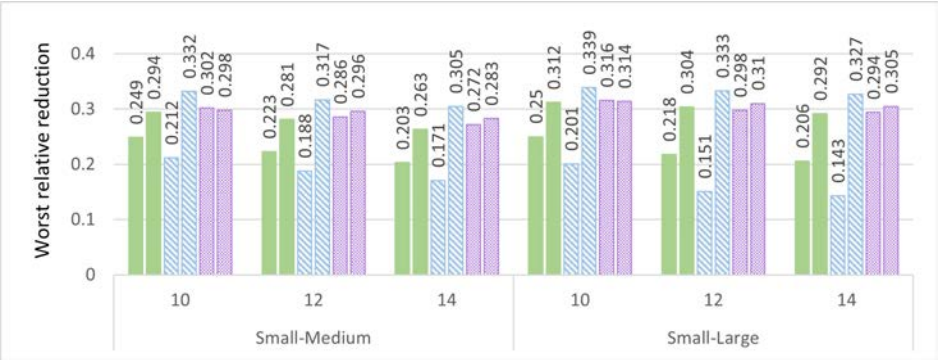
First we focus our evaluation in homogeneous mission scenarios, where all drones are assigned the same mission and follow the same pre-specified paths. We perform experiments for all mission types and fleets with 10, 12 and 14 drones. The results are shown in Figure 3.4a. The bars show the *worst* relative makespan reduction over all drones achieved by the RR, CB and EB policy for each mission type.

The RR policy manages to reduce the default mission times by 18.5% up to 24.9% depending on the mission type and number of drones competing for the server resources. The CB policy consistently outperforms RR, achieving an *additional* reduction of $3.3\%, 3.3\%$ and 2.8% averaged over all mission types for 10, 12 and 14 drones, respectively. In absolute terms, the mission time is *further* reduced vs RR by about 0.5 min, 1 min and 1.5 min for the small, medium and large missions respectively, for all fleet sizes. The absolute total reduction of the mission time ranges from about 3.5 - 4.5 min, 7.5 - 9 min and 11.5 - 13.5 min, for the small, medium and large missions, respectively. The EB policy achieves similar results to CB.

Further, we evaluate RR, CB and EB for heterogeneous mission scenarios, where half of the drone fleet follows the small mission and the other half follows the medium mission (small-medium experiments) or the large mission (small-large experiments). The results are presented in Figure 3.4b; two bars are shown for each policy, the first (shorter) bar for drones with the small mission and the second (higher) bar for drones with the medium or large mission. As it can be seen, RR and CB do not produce balanced results, having a clear tendency to favor drones with larger paths. In fact, CB is even more unfair than RR in this respect and actually achieves an overall worse result (shortest bar in each pair) in light of the optimization objective as per Equation 3.11. In contrast, the EB policy manages to smoothly balance the makespan reduction over different mission types in all cases, achieving a far better overall result. Compared to RR, EB *further* decreases the worst relative makespan by an *additional* 5.3 - 8.8% (1 - 1.5 min) for the drones with the small missions, also leading to an *improved* reduction of 0.2 - 2.0% (up to 0.5 min) for the drones with the larger (medium or large)



(a) Homogeneous paths



(b) Heterogeneous paths

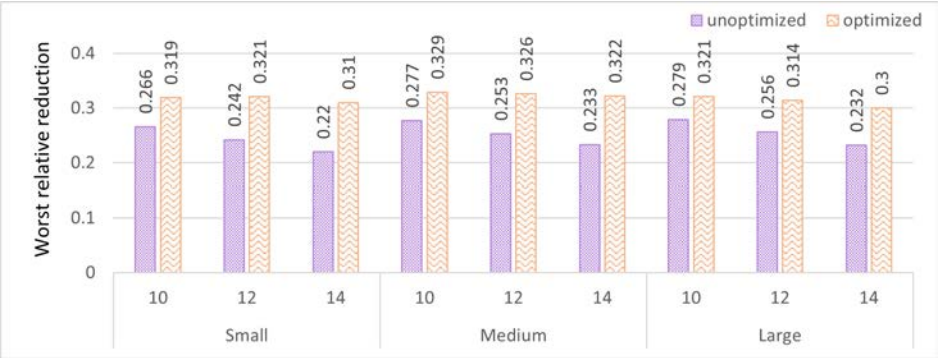
Figure 3.4: Worst relative reduction of the mission time over the default, achieved by the RR, CB and EB policies.

missions.

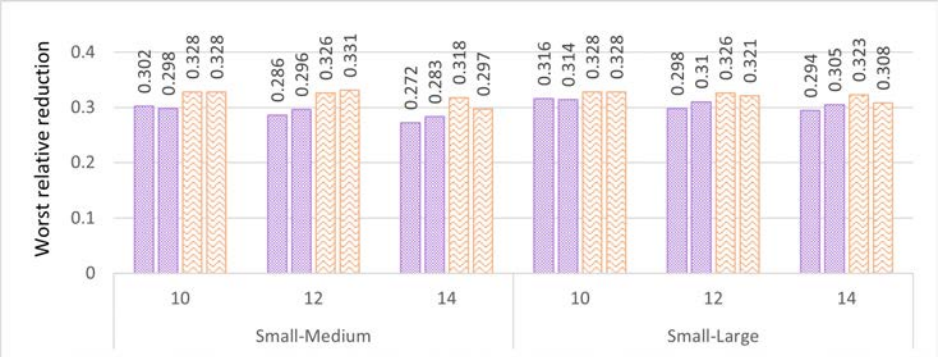
Evaluation of path reversal optimization

In another set of experiments, we evaluate the path reversal optimization for the EB selection policy. The results for the homogeneous and heterogeneous mission experiments are shown in Figure 3.5. Clearly, this optimization achieves significant extra reduction of the makespan, beyond what is already achieved by the EB policy without path reversal, as it manages to reduce the contention between drones.

The *additional* reduction is 4.2% - 9.0% for the homogeneous mission scenarios, which in absolute terms amounts to about 1 - 1.5 min for the small missions, 1.5 - 3 min for the medium missions, and 2 - 3.5 min for the large missions. The path reversal optimization is also beneficial in the heterogeneous mission scenarios, leading to an improvement vs "plain" EB of 1.2% - 4.6% (up to 0.5 min) for the drones with the small missions and 0.3% - 3.5% (up to 1 min) for the drones with the larger missions. The reason for higher improvement in the homogeneous missions is that there is consistent and higher contention among all drones, which



(a) Homogeneous paths



(b) Heterogeneous paths

Figure 3.5: Worst relative reduction over default mission, achieved by the unoptimized and optimized version of the EB policy.

is relaxed by letting some drones follow reversed paths. The maximum reduction achieved is 33% and the maximum absolute gain is 16 min.

3.6.4 Heterogeneous drones

In this set of experiments we assume that 24 heterogeneous drones operate in the mission area shown in Figure 3.3, performing the same sensing and processing task at each point of interest. The heterogeneity between drones is introduced as follows. On the one hand, we let some drones perform the computation 20% faster than what was measured with the RPi, assuming that these drones feature a faster computing platform. On the other hand, we assume that some drones feature stronger motors or lighter payload, which allows them to travel faster covering the same distance in 20% less time than what is estimated in the above flight model. Combining these two heterogeneity aspects, we consider four different types of drones, shown in Table 3.2. More precisely, drones of type A and C can compute faster, while drones of type A and B can fly faster. Note that type A drones, have the best settings in terms of both computation and travel speed, while type D drones have the worst settings.

Table 3.2: Drone heterogeneity in terms of computing and flight capability.

	Type A	Type B	Type C	Type D
Travel Speed Factor	0.8	0.8	1	1
Local Computation Factor	0.8	1	0.8	1

Table 3.3: Settings for the parameters guiding the random search of s-EPPOS.

Parameter	value
$moves^{low}$, $moves^{high}$	1, $\frac{M}{3}$
$first\_occ^{low}$, $first\_occ^{high}$	1, 5
$last\_occ^{low}$, $last\_occ^{high}$	1, 5
$prob_{swap}$	0.1
$swaps^{low}$, $swaps^{high}$	$\frac{M}{2}$, M

For the sake of symmetry, we equally divide drones among the different types A, B, C and D, in other words, we have 6 drones of each type.

Here, we also employ a simplified version of the EPPOS algorithm described in Section 3.5, referred to as s-EPPOS, assuming that the energy is sufficient for the whole mission. This assumption is also made by the greedy heuristic. The s-EPPOS is set with the parameters given in Table 3.3, while the number of top level $iterations$ is set to 400.

Results - same missions

In a first set of experiments, we investigate the scenario where all of the drones need to visit the same points of interest, for the small, medium and large missions. Figure 3.6 shows the results in the form of boxplots. In each plot, the first three boxes correspond to the different policies of the greedy algorithm described in Section 3.4, while the fourth boxplot captures the results of the s-EPPOS heuristic. The Y axis denotes the relative reduction vs the default makespan. On each box we mark the relative reduction of each individual drone using different marker shapes for the four different drone types. Also, on the bottom of each box we give the worst relative reduction (smallest value among all drones), which is what we wish to maximize as per Equation 3.11. The results for the small, medium and large mission is shown Figure 3.6(a), Figure 3.6(b) and Figure 3.6(c), respectively.

We observe that the best results are found by the EB policy of the greedy algorithm and the s-EPPOS meta-heuristic. On the one hand, EB considers the contention between the drones and the expected total relative reduction that each drone will achieve, thereby finding a bal-

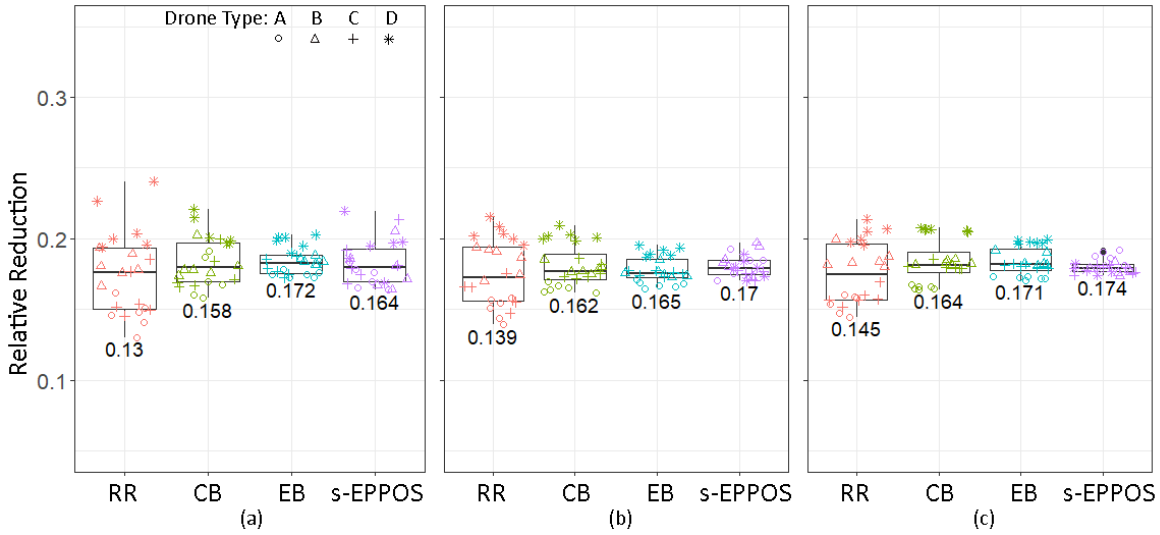


Figure 3.6: Same missions. (a) small, (b) medium and (c) large.

anced and fair schedule. EB finds schedules with 16.9% average worst relative reduction over all missions, while the average standard deviation between the relative reduction of each drone, is 0.95%. On the other hand, s-EPPOS iteratively explores multiple randomized solutions, converging to a good and fair plan. The average worst relative reduction achieved by s-EPPOS is also 16.9%, while the average standard deviation over all missions is 0.94%, practically the same as EB.

The CB greedy policy finds a little worse results than EB, achieving an average worst relative reduction of just 16.1%. The reason is that CB does not consider the contention between drones, making it harder to share the edge resources in a fair way. Due to the heterogeneity among drones, some complete their mission faster than others. This means that some drones will be exposed to less contention in total than others. The lack of fairness can also be observed by the increased standard deviation between the drones which for CB is 1.55% on average over all missions.

Finally the RR heuristic finds the worst and most unfair schedules vs all other approaches. This is because it blindly gives edge resources to each drone without considering the resulting relative reduction or contention. The worst relative reduction and the standard deviation are $13.8\%, 2.41\%$ respectively, averaged over all missions. Notably, s-EPPOS outperforms RR by $3.4\%, 3.1\%$ and 2.9% for the small medium and large mission, respectively.

By looking at the individual drone markers on the boxplots, we can observe that in all the versions of the greedy algorithm (RR, CB and EB), the worst improvement is achieved for the type A drones. This is reasonable since these drones have the best settings, making

any optimizations less important. On the one hand, the shorter local computation time makes the time difference vs offloading smaller. On the other hand, due to the shorter travel times, type A drones are the first to complete their missions, leaving more room (less contention) for the rest of the drones, while they are exposed to the highest contention (with all other drones) during their entire mission. For the opposite reasons, the best results are achieved for type D drones. The type C drones achieve worst improvements compared to type B and D. This is because they have fast hardware so offloading becomes less significant for the overall mission time.

The s-EPPOS meta-heuristic, does not follow this pattern. The reason is the way this algorithm operates. Namely, s-EPPOS finds a good schedule after multiple iterations of randomly shuffling the decision order of the drones. In these mutations, the algorithm does not take into account the type of the drone and (in part) does not even consider which drone actually has the worst expected relative reduction of the mission time. As a consequence of this randomized approach, s-EPPOS does not favor a particular type of drone.

Finally, an interesting observation is that for the small missions, shown in Figure 3.6(a), EB outperforms s-EPPOS even if slightly by 0.8%. The reason is that EB builds a schedule based on the estimated future relative reduction of all drones. If the estimation is close to the real relative reduction that a drone will achieve, then this approach is more appropriate/accurate than building a solution randomly, even when the random procedure is repeated for a number of iterations. However, this is not the case for medium and larger missions shown in Figure 3.6(b) and Figure 3.6(c), where s-EPPOS outperforms EB by 0.5% and 0.3%. This is because the estimation of the expected reduction of mission time done by EB becomes less accurate as the length of the mission increases as in these cases the future is less predictable.

Results - different missions

In a second set of experiments, the drones need to visit different points of interest according to the three cases shown in Figure 3.3. We explore three scenarios. In the first scenario, half of the drones are assigned the small mission and the other half follow the medium mission. In the second scenario, half of the drones are assigned the small mission and the other half follow the large mission. In the third scenario, half of the drones are assigned the medium mission and the other half follow the large mission. For the sake of symmetry, we divide the drones of each type so that half of them are assigned the smaller mission (small or medium)

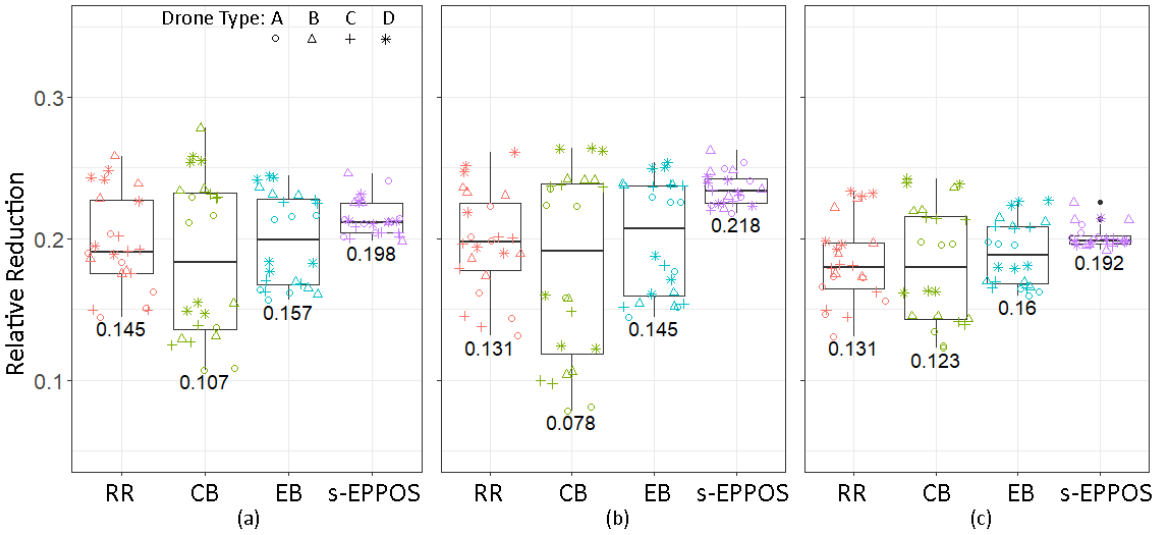


Figure 3.7: Different missions. (a) small-medium, (b) small-large and (c) medium-large.

while the other half follow the larger (medium or large). For example, 3 type A drones follow the smaller mission and the other 3 follow the larger mission. Same applies for the rest of the drone types. Figure 3.7 shows the results for each scenario in the form of boxplots.

First of all, we observe that the standard deviation of the relative reduction achieved by each drone is greater for the RR, CB and even EB versions of the greedy algorithm, compared to the previous set of experiments. More precisely, the deviation becomes 3.33% (vs 0.95%), 5.47% (vs 1.55%) and 3.2% (vs 2.41%) for RR, CB and EB, respectively.

Interestingly, the worse results are achieved by the CB policy rather than RR (as in the previous experiments). This is because CB does not consider the contention, as already discussed in the previous subsection. The impact becomes even more significant here because the length of the missions differ. Since CB myopically considers the current reduction in the mission time of each drone, it gives priority to drones with longer paths because the time saved from an offloading decision is not as significant as the time saved from an offloading when the path is shorter. This bias exists in all three scenarios, as can be confirmed by checking the points that represent the different drone types. We can see that half of the drones of a given type achieve much better relative reduction in the mission time vs the other half. As a result, CB becomes more unfair than RR.

The EB policy achieves better results vs the other two policies, outperforming RR by $1.2\%, 1.4\%$ and 2.9% for the small-medium, the small-large and the medium-large scenarios respectively. Still, looking at the data points for the same drone type, we observe a similar pattern as in CB: there is a separation between half of the drones of each type, with better

results being achieved for the drones following the longer vs the shorter path. This is because the estimations of EB regarding the total relative reduction of each drone are more uncertain when the path is longer, underestimating the reduction that can be achieved for the drones with the longer path in the future, after the drones with the shorter path will have finished their missions. As a result, the EB policy is slightly more favorable for the drones with the larger missions.

Notably, the difference between s-EPPOS vs EB becomes far more significant in this set of experiments, 4.1%, 7.3% and 3.2% for the small-medium, the small-large and the medium-large scenarios respectively. This is because EB fails to estimate the total relative reduction of each drone accurately, leading to unfair schedules vs those produced by s-EPPOS. Furthermore, s-EPPOS achieves a more balanced improvement for all drones at much smaller standard deviation of 1.1% on average over all scenarios, almost $\frac{1}{3}$ of the deviation of the EB greedy policy. Also, like in the previous experiments, s-EPPOS does not favor a particular type of drone, whereas RR, CB and EB achieve better improvements for drones of type B and D, for the reasons explained in the previous subsection.

3.7 Evaluation of EPPOS Considering Energy Constraints

We evaluate the EPPOS algorithm through simulation experiments for scenarios where the autonomy of the drones do not suffice for the whole mission. The goal of our evaluation is to demonstrate the performance, robustness and fairness of EPPOS for scenarios with various battery switching time delays at the depot station and various operational autonomy of the drones.

3.7.1 Topology and missions

The missions are conducted in the same 21×21 grid shown in Figure 3.3. The depot location as well as the edge server locations are the same as in the previous experiments.

We perform our experiments for randomly generated missions in the above topology, by randomly picking for each drone from 60 up to 80 points of interest (each drone is assigned a different random mission). The randomly generated points of interest are fed into a typical TSP algorithm to produce the initial paths (P^{init}) for the drones, which, in turn, are handed over to the EPPOS algorithm in order to produce the optimized drone paths and offloading

plan to the shared edge infrastructure.

3.7.2 Drone and server model parameters

The setup is similar to the evaluation setup in Section 3.6, considering the same edge server hardware and same embedded platform for the drone's onboard computer. As a result, we reuse the sensing delay, as well as the processing times of the data, as measured (in both the server and the embedded device) in Section 3.6.

For the horizontal movement of the drone, we assume steady acceleration 0.8 m/s^2 when departing from a waypoint until the drone reaches the operational speed of 4 m/s, and steady deceleration 1.6 m/s^2 as the drone approaches and stops at the next waypoint. So, for example, the distance of 20 meters between two neighboring points in the above topology (Figure 3.3), is covered in about 9 seconds. Also, we assume that the drone needs 5 seconds for a vertical take-off from the depot to the operational flight altitude of 10 meters, and 20 seconds to perform vertical landing at the depot. In this set of experiments we assume a faster acceleration/deceleration for the flying behavior of the drones than in Section 3.6. This is so we can decrease the time to move from one point to another, increasing the contention in the servers (because drones will require offloading more frequently), while still keeping the flight model realistic as per ArduPilot [86].

The autonomy of the drone is the total amount of time it can remain in the air based on its battery capacity, including the time needed to take-off from the depot, fly from one point of interest to the next, hover over a point of interest to perform the required sensing and data processing, and to land at the depot. As already discussed in the Section 3.3, drones always leave the depot with their batteries fully charged. We assume that drones consume the same amount of energy when hovering, flying, taking-off and landing. This assumption is quite realistic for small drones given that the above flight model includes moderate acceleration/deceleration values.

3.7.3 Settings for EPPOS parameters

The settings for the parameters of the random search heuristic of EPPOS are given in Table 3.4. We arrived at those values empirically, after a number of exploratory runs. With these settings, the algorithm converges quickly to a good solution. More specifically, in practically all the exploratory runs, convergence was reached long before 400 iterations – additional

Parameter	value
$moves^{low}$, $moves^{high}$	1, $\frac{M}{3}$
$first\_occ^{low}$, $first\_occ^{high}$	1, 5
$last\_occ^{low}$, $last\_occ^{high}$	1, 5
$prob_{swap}$	0.1
$swaps^{low}$, $swaps^{high}$	$\frac{M}{2}$, M
$prob_{pTchng}$	0.6
$pTchng^{low}$, $pTchng^{high}$	0.7, 1.3

Table 3.4: Settings for the parameters guiding the random search of EPPOS.

iterations had a negligible effect on the quality of the solution yielding marginal or no improvement. Thus, in the experiments presented here, we set the $iterations$ parameter to 400.

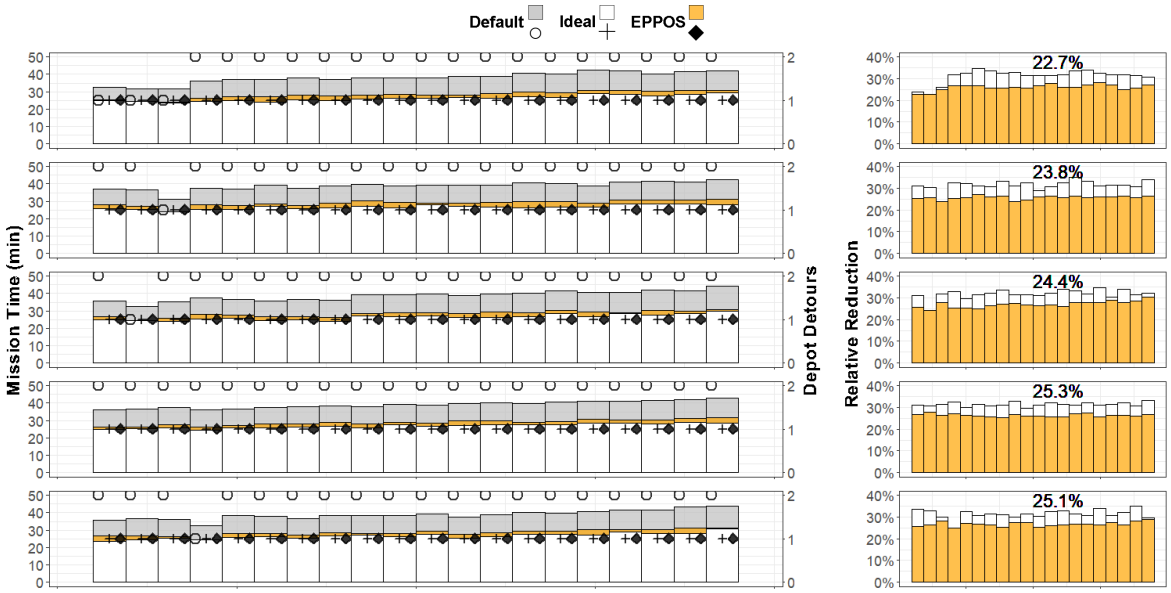
We note that the time that is needed to perform these iterations is acceptable for an offline algorithm, roughly 1 minute. This time was measured on a commodity laptop running Windows10, with i7-8550U CPU and 8GB of RAM, with an implementation of the EPPOS algorithm in Python3.

3.7.4 Presentation of results

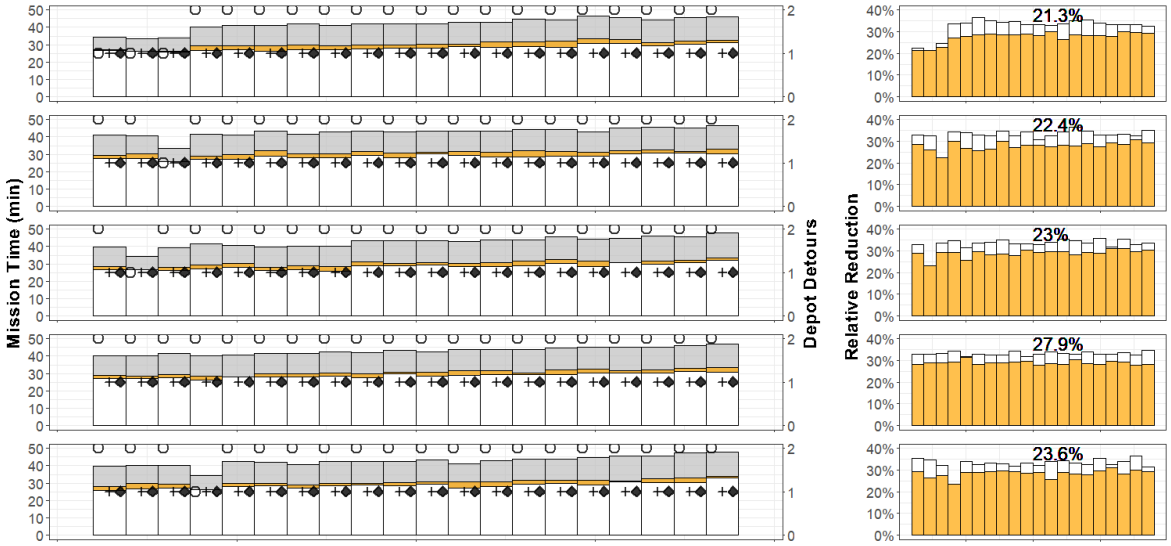
In the sequel, we report the results of the experiments that are performed for 20 drones operating concurrently in the mission area, where we vary (i) the battery switching time at the depot and (ii) the autonomy of the drones.

As a reference for the results produced by EPPOS, we use a deterministic algorithm that optimizes the initial paths (P^{init}) under the assumption that each drone can immediately offload its computation, with zero waiting time, to one of the servers that are in range of the respective points of interest. This algorithm works in a similar way to the part of the AdjustPaths() function (see Algorithm 8) where intermediate depot detours are added to make the path safe, without the path reversal logic. Note, however, that since the algorithm assumes perfect offloading, the result corresponds to a *potentially infeasible* lower bound for the mission time (and upper bound for the maximum relative reduction vs the default mission time), for the ideal case where there is no contention between the drones that operate concurrently in the mission area.

For each experiment scenario, we report the outcome of 5 different sets of randomly generated missions (every set consists of 20 missions, one mission per drone). Given that the EPPOS algorithm is itself randomized, we run it five times for each mission set and report



(a) Battery switching delay is 3 minutes.



(b) Battery switching delay is 5 minutes.

Figure 3.8: Different battery switching delays (autonomy is 15 min).

the median. The results are presented via two barplots placed next to each other (one bar per drone, one row per mission set). In the left plot, the bars show the mission time achieved by EPPOS vs the default and ideal mission time for each drone (left Y-axis). We also indicate the number of depot detours that are introduced for each drone (right Y-axis). In the right plot, we show the relative reduction for EPPOS as well as for the ideal case assuming no contention vs the default mission time. The percentage on top of the plot indicates the worst relative reduction that is achieved by EPPOS over all drones (the optimization objective).

3.7.5 Experiments for different battery switching delays

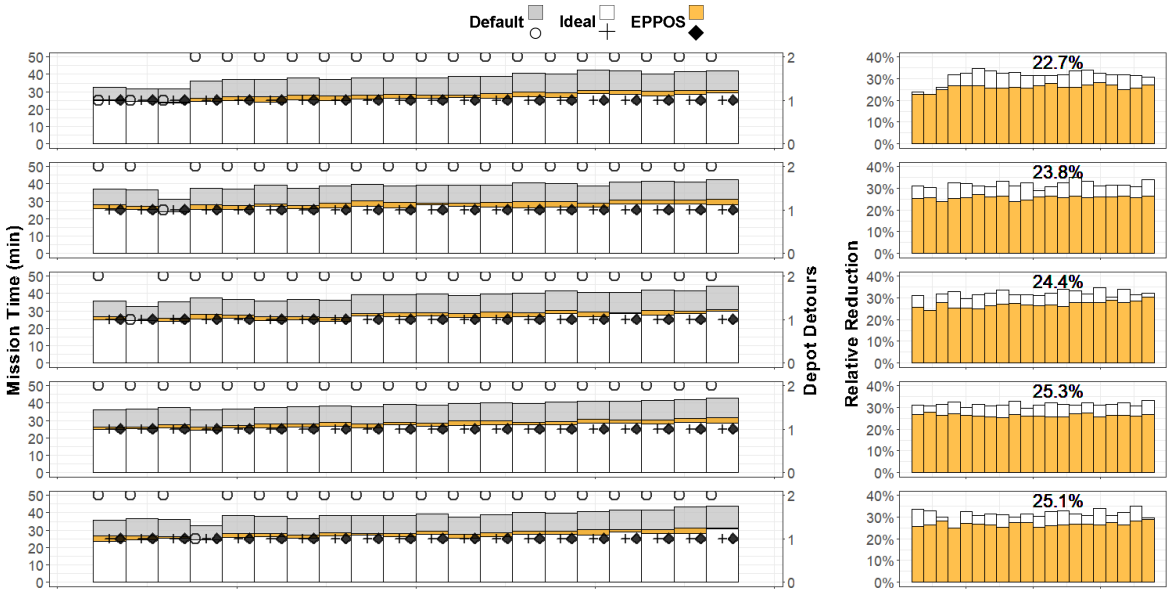
In a first set of experiments we study the performance of the EPPOS algorithm for battery switching delays of 3 minutes and 5 minutes, for an autonomy of 15 minutes. The results are shown in Figure 3.8. Averaged over all five randomly generated path sets, the worst relative reduction of the mission time across all drones is 24.3% and 23.6% for a battery switching delay of 3 and 5 minutes, respectively. Compared to the ideal case of zero contention, the results of EPPOS are worse by merely 0.89% and 0.75%.

The difference between EPPOS and the ideal case becomes less significant for an increasing battery switching delay. The reason is twofold. Firstly, larger battery switching delays reduce contention as every drone that makes a depot detour spends more time at the depot and as a result stays out of competition for the shared edge resources for a longer period. Secondly, when the battery switching delay is large and a depot detour is unavoidable in order for a drone to safely complete its mission, the time that can be saved thanks to offloading becomes less significant for the total mission time.

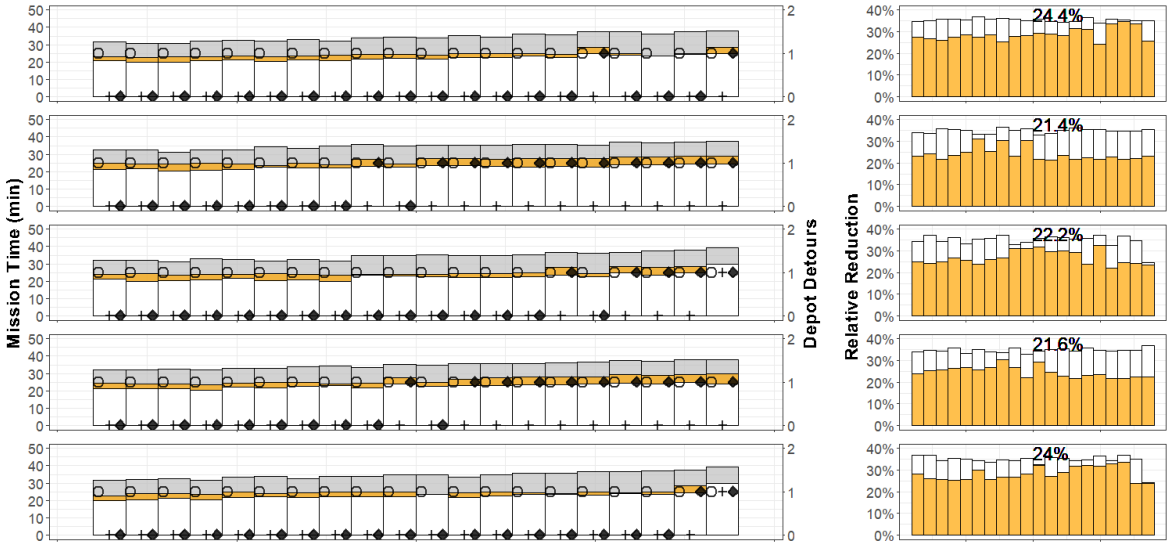
3.7.6 Experiments for different degrees of autonomy

In a second set of experiments, we keep the battery switching delay to 3 minutes and vary the autonomy of the drones. In addition to the autonomy of 15 minutes, we investigate scenarios with an autonomy of 25 minutes. The results are shown in Figure 3.9. Note that Figure 3.9a is identical to Figure 3.8a; it is shown again here to enable a direct visual comparison with the larger autonomy scenario.

Averaged over all five randomly generated path sets, EPPOS achieves a worst relative reduction of the mission time across all drones, of 24.3% and 22.7% for an autonomy of 15 and 25 minutes. The result is 0.89% and respectively 7% worse than the ideal case. Note that the difference of EPPOS vs the ideal increases for the scenario with a larger autonomy. The reason is that the ideal mission planning algorithm assumes zero contention and can fully exploit the larger autonomy to complete the mission with fewer depot detours. However this does not mean that theses savings are entirely realistic given that the actual contention between drones is non negligible. This, in turn, may lead to the loss of some offloading opportunities and force EPPOS to introduce additional depot detours in order for the drones to be able to complete their mission. It is also possible that the algorithm fails to escape a local optimum in the solution space.



(a) Drone autonomy is 15 minutes.



(b) Drone autonomy is 25 minutes.

Figure 3.9: Different degrees of autonomy (battery switching delay is 3 minutes).

An important observation, is that EPPOS manages to produce balanced plans where all drones achieve a similar reduction of their default mission time, which is particularly valuable in light of fairness. Note that, as a side-effect of the optimization criterion, EPPOS tends to maximize the reduction for the drones that have lower improvement potential (shortest white bars) before further improving the mission times of others.

3.8 Offloading Protocol

Now we focus on scenarios where the drones may deviate from the initial schedule due to uncertain flight and processing times. We describe an online decentralized approach to exploit such uncertainties and optimize the schedule.

Each drone d_m executes its mission based on the plan $(P_m S_m, W_m)$ that is generated by the offline algorithm. However, when the drone reaches a point of interest where the plan is to offload its computation to the edge ($S_m[i] \neq 0$), it runs a protocol with all edge servers in range to decide whether to offload its computation to one of them. The protocol is illustrated in Figure 3.10, for indicative offloading scenarios.

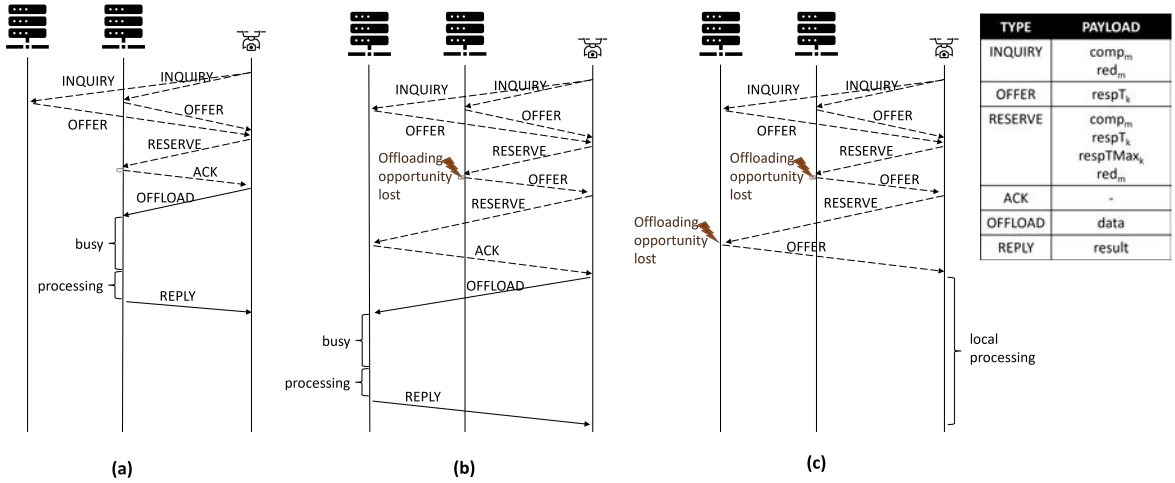


Figure 3.10: Indicative scenarios for offloading the drone's computation to the edge. The messages of the ad-hoc negotiation protocol are shown with dashed arrows. Note that these have fixed sizes in the order of a few bytes, independently of the computation to be offloaded. The solid arrows show the actual offloading interaction (if the drone decides to engage the edge server).

As a first step, the drone sends an INQUIRY message to all servers in range, which includes the type of the computation at hand $comp_m$ and the expected relative reduction red_m vs the default mission time. In turn, each server s_k replies with an OFFER message, which indicates the estimated response time $respT_k$. This is calculated based on the processing time on the server's hardware platform $procT(comp_m, hw_k)$, the current load of the server as well as any pending offloading requests in a local priority queue. More specifically, if d_m has a worse expected reduction red_m compared to other pending requests, the server will give priority to it (in line with the optimization objective). Note, however, that the offer made by the server is non-binding as no actual resource allocation takes place and no entry is added in the

priority queue at that point.

When the drone receives the offers from the servers, it picks the server s_k that replied with the best offer (shortest $respT_k$), and calculates the expected end-to-end computation time $compT(m,k)$ as per Equation 3.2 by replacing $procT(comp_m, hw_k)$ with $respT_k$. If this is beneficial vs the local execution of the computation $procT(comp_m, hw_m)$, it sends to s_k a RESERVE message to make an actual reservation for the offered response time $respT_k$. In this message, the drone also includes an upper bound for the response it is willing to accept (in case the server needs to prioritize other drones, whose requests may arrive in the near future). This is calculated as $respTMax_k = max(respT_k, visitT_m^i - senseT_m)$, where $visit_m^i - senseT_m$ is the expected offloading time in the offline plan at $P_m[i]$. If the server can meet $respT_k$, it makes the necessary resource allocation, adds the request of d_m in the priority queue, and replies with an ACK message, which is a hard commitment to perform the computation by the indicated maximum response time. In this case, shown in Figure 3.10(a), the drone offloads the computation to s_k and waits for the response before it continues its mission.

However, at the time s_k receives the RESERVE message from d_m, it may have already committed its resources to another drone that run the same protocol concurrently to d_m. In this case, the server replies with another OFFER message and an updated $respT_k$. Based on the new offer, the drone repeats the selection process. As a result, it may decide for another edge server to which it will send a RESERVE message. If that server replies with an ACK, the drone will offload its computation to it, as shown in Figure 3.10(b).

The drone repeats the inquiry-reservation procedure until a beneficial offloading option is successfully reserved, or the best offer received is not beneficial vs local computation. In the latter case, shown in Figure 3.10(c), the drone decides to perform the computation locally. Note that, in the worst case, d_m may perform up to $\frac{compT(m,0)}{avg_{m' \neq m=1}^{M} compT(m', hw_k)}$ consecutive reservation attempts, for each server s_k in range of the current waypoint, before it decides to perform the computation locally.

Algorithm 10 shows the high level mission execution logic of the drone. Each drone starts following the path that was generated by the offline algorithm. However, before moving from the current location $P_m[i]$ to the next point of interest $P_m[i+1] \in \mathcal{V}_m$, a check is made to ensure that it is safe to make the trip and perform the required sensing and processing at the point of interest, based on the worst possible scenario where the drone experiences the maximum

Algorithm 10 Drone runtime logic

function DRONESIDE($P_m, S_m, W_m, missionT_m^{def}$)
  $missionT_m \leftarrow$ calcMissionT$(P_m, S_m, W_m, 1)$
  $red_m \leftarrow$ calcReduction$(missionT_m^{def}, missionT_m)$
  $t \leftarrow 0$
  $i \leftarrow 1$
  $e_{rem} \leftarrow E_m^{max}$
  while $i <$ len(P_m) **do**
    if $P[i+i] \in \mathcal{V}_m$ **then** ▷ going to a point of interest
      $e_{fly} \leftarrow \alpha_m \times flyMaxT_m^{i,i+1}$
      $e_{visit} \leftarrow \beta_m \times (senseT_m + compT(m, 0))$
      if $e_{rem} \leq e_{fly} + e_{visit}$ **then**
        insertDetour(P_m, i, dep_m)
      end if
    end if
    $flyT_m^{i,i+1} \leftarrow GOTO(P_m[i+1])$
    $e_{rem} \leftarrow e_{rem} - \alpha_m \times flyT_m^{i,i+1}$
    $t \leftarrow t + flyT_m^{i,i+1}$
    $i \leftarrow i + 1$
    if $P_m[i] \in \mathcal{V}_m$ **then** ▷ at a point of interest
      $t_{beg} \leftarrow$ getTime()
      $data \leftarrow SENSE()$
      $k \leftarrow 0$
      if $S_m[i] \neq 0$ **then**
        $srv \leftarrow$ getServersInRange$(P_m[i])$
        $k \leftarrow$ protocol$(srv, comp_m, red_m)$
      end if
      if $k \neq 0$ **then**
        $reply \leftarrow OFFLOAD(s_k, data)$
      else
        $reply \leftarrow PROCESS(data)$
      end if
      $t_{end} \leftarrow$ getTime()
      $visitT_m^i \leftarrow t_{end} - t_{beg}$
      $e_{rem} \leftarrow e_{rem} - \beta_m \times visitT_m^i$
    else
      $visitT_m^i \leftarrow SWITHBATTERY()$
      $e_{rem} \leftarrow E_m^{max}$
    end if
    $t \leftarrow t + visitT_m^i$
    adjustPath(P_m, e_{rem})
    $missionT_m \leftarrow t +$ calcMissionT(P_m, S_m, W_m, i)
    $red_m \leftarrow$ calcReduction$(missionT_m^{def}, missionT_m)$
  end while
end function

flight time and ends-up performing the computation locally. If the remaining energy of the drone is not sufficient, a depot detour is added at that point (via the insertDetour() function). Note that this check is needed even though the offline algorithm produces the plan based on the worst-case flight times. Since the drone may not fully respect the offloading actions of the offline plan, the contention on the edge servers may lead to larger delays, rendering the path unsafe. As an exception, the check is not needed when the drone is heading directly to the depot ($P_m[i+1] = dep_m$) because this hop is always guaranteed to be safe.

If the hop is safe, the drone moves to $P[i+1]$ and (if this is a point of interest rather than the depot) performs the required sensing task. Then, if $S_m[i] \neq 0$, it runs the protocol described above. Function protocol() abstracts the communication with the nearby edge servers, returning the identifier of the server where the computation should be offloaded. In case no beneficial offloading option is found, the function returns 0, indicating that the computation should be performed locally. Note that the computation is always performed locally if $S_m[i] = 0$ according to the offline plan.

This procedure continues until the drone visits the last point of interest and returns back to the depot. Notably, since the actual flight times are most likely to be shorter than the worst-case assumed in the offline plan, the drone's mission time and energy consumption is likely to decrease vs the offline plan. To exploit this gain, the drone checks whether it can safely optimize its path by postponing or even eliminating a depot detour (adjustPath() function). If, however, there is energy loss due to increased contention, an already planned depot detour might be moved earlier in the path or even a new one may be inserted. The expected mission time and the reduction vs the default mission time is also updated after each hop. To this end, the drone tracks the time that has elapsed up to the current point $P[i]$, and adds to it the expected mission time from that point onward as per the offline plan, computed via function calcMissionT(), in the spirit of Equation 3.9, but starting from the indicated mission point rather than the beginning of the mission. The reduction vs the default mission time is then calculated via function calcReduction() as per Equation 3.10. This is done even if the drone's path has not changed, as the mission time can be reduced merely due to the shorter flight times.

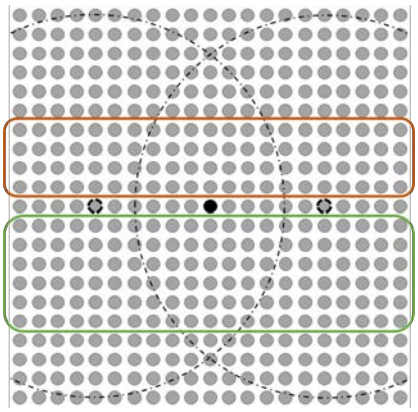


Figure 3.11: Mission topology.

3.9 Evaluation

We evaluate our approach through simulations. We start by presenting the experimental setup and the key simulation parameters. Then, we discuss the results of our experiments.

3.9.1 Topology and missions

We conduct our experiments in the same $21x21$ grid, shown in Figure 3.3, also given in Figure 3.11 for convenience. The neighbor points are 20 meters apart. The edge servers, featuring antennas with 200 meters range, are denoted with the black borders.

We run experiments for two mission types, denoted via the rounded rectangles. In the small mission (orange) each drone must visit 84 points of interest, whereas in the large mission (green) each drone must visit 126 points of interest. We assume a common depot station at the center of the grid (black node), which can be used concurrently by all drones.

3.9.2 Drones

We run experiments for 20 drones with the same flight properties, energy capacity and hardware platform. Also, we assume that every drone performs the same sensing task and needs to perform the same type of computation on the data. For the experiments we use the same flight behavior as in Section 3.7. The offline algorithm produces mission plans based on this flight behavior with the respective flying times assumed to be the worst case values $flyMaxT_m^{i,i+1}$. In the simulated mission execution, the flight times follow a uniform random distribution with the lower bound set to $flyMinT_m^{i,i+1} = (1-u) \times flyMaxT_m^{i,i+1}$ where u is the uncertainty level. We run experiments for $u = 20\%$ and 30%. We store the randomly

generated flight times between every two points in a file, and use these values in the runs for the different approaches.

Drones are assumed to have an autonomy of 15 minutes, whether cruising or hovering. The battery switching time $depT_m$ at the depot is set at 3 minutes and is assumed to be correctly estimated (the value used in the offline algorithm is the same as in the simulated execution).

Similarly to the setup in Section 3.7, the drones take pictures, with $senseT$ equal to 1 second, while the processing time using their onboard computer is 10 seconds.

3.9.3 Edge servers

We assume same server parameters as in Section 3.7. However in this section we analyze the different overheads for the offloading in more detail.

The bandwidth bw between the drones and the edge servers is set to 50Mbps, which is realistic for WiFi. Then, the time that is required to send the offloading request (and image) to the server is about 150 milliseconds. Given that the rest of the offloading protocol messages are very small, we assume that the respective transmission time is dominated by the network latency, set to 10 milliseconds. In total, the time $compT(m, k), k \neq 0$ it takes for the drone to receive the results when the computation is offloaded to the edge is about 2 seconds (without any additional delays due to the protocol negotiation phase or possible contention on the server). We have verified that this is very close to the actual offloading of the computation between the RPi and the laptop over WiFi. This value (in addition to any waiting times) is used for the generation of the offline plan as well as in the simulated mission execution. Notably, we do not model the physical medium or the effects of mobility and interference to the quality of the wireless link.

3.9.4 Benchmarks

As a benchmark for the results that are achieved by our approach, we use different alternatives, briefly discussed below. The EPPOS algorithm (Section 3.5) is used for all the plans produced offline.

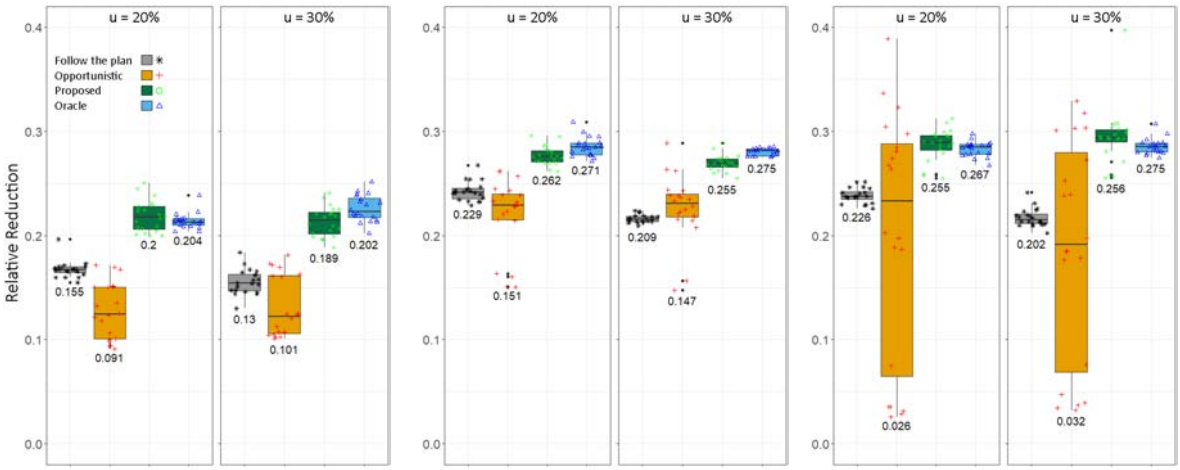
Follow the plan. The drone follows the plan that was produced offline. More precisely, when d_m arrives at $P_m[i] \in \mathcal{V}_m$ and completes the sensing task at that location, it offloads the computation to $S_m[i]$ according to plan (waiting as needed before offloading). The edge

servers process requests based on a FCFS policy without taking into account the drone's expected reduction. Note that, in this case, there is no negotiation phase and the drone always offloads to the server specified in the plan or performs the computation locally if $S_m[i] = 0$.

Opportunistic offloading. The drone follows the path of the offline plan. However, when d_m arrives at $P_m[i] \in \mathcal{V}_m$ and completes the sensing task, it runs the protocol and takes its offloading decision irrespective of the planned $S_m[i]$ and $W_m[i]$. Notably, the drone may decide to offload even though the plan is to perform the computation locally ($S_m[i] = 0$). As above, the edge servers process requests based on a strict FCFS policy without taking into account the drone's expected reduction. As in the proposed approach, the drone exploits any gains to postpone or eliminate depot detours.

Oracle. The mission plan is produced by the offline algorithm, based on the actual time that will be needed in the simulated mission execution to fly between two points $P_m[i]$ and $P_m[i+1]$ in the mission plan (rather than the respective worst-case values $flyMaxT_m^{i,i+1}$). The drone simply follows the path and offloading schedule of the offline plan. This basically serves as an upper bound for the result that could be achieved by the other approaches.

3.9.5 Results and discussion



(a) Small mission with 20 drones.

(b) Large mission with 20 drones.

(c) Small and large mission 10 drones each.

Figure 3.12: Relative mission reduction achieved by the proposed approach vs the three benchmarks.

Figure 3.12 illustrates the results obtained for a particular set of randomly generated flight times. All other runs with different random values are similar. Figures 3.12a and 3.12b show

the results for the case where all drones follow the same mission, small and large respectively. Figure 3.12c shows the results for the scenario where where half of the drones follow the small mission and the other half follow the large one. The left part of each figure is for $u = 20\%$, the right part is for $u = 30\%$. The results achieved by each approach (proposed vs the three benchmarks) are illustrated as in the form of a boxplot, where the solid area of the box denotes the borderline to the upper and lower quartiles, the horizontal line within the area marks the median, the vertical line denotes the lower and upper quartiles, and the outliers are shown as separate dots. The markers show the relative mission time reduction achieved by each individual drone. The lowest marker (drone with the worst relative reduction) is the reference for the target optimization objective, as per Equation 3.11. The corresponding value is printed directly below the respective marker in order for this metric to be more clearly visible.

The proposed approach outperforms follow-the-plan in all cases, improving the worst relative reduction by about 3.6% and 5.4% averaged over all experiments with uncertainty 20% and 30%, respectively. The difference becomes greater with increased uncertainty. This is expected since the latter does not take advantage of the shorter flight times that may occur during the mission, whereas the proposed approach exploits such gains to optimize the path of the drone. What is interesting in this respect, is that the proposed approach performs the same number of depot detours as the follow-the-plan approach. However, these depot detours are shifted to a more appropriate point in time during the mission, thereby reducing the total flight time vs the follow-the-plan approach, by 1.2% and 1.3% for uncertainty 20% and 30%, respectively.

Notably, the results achieved by the proposed approach are, on average, merely 0.8% and 1.2% worse vs the oracle algorithm. Even though in some cases the proposed approach manages greater relative reduction for most drones (e.g., Figure 3.12c), the oracle achieves a better worst-case relative reduction over all drones, which is the optimization objective.

It is important to observe that simple opportunistic offloading consistently leads to (far) worse results with respect to the optimization objective, even compared to the follow-the-plan approach, especially in the mixed mission experiments (Figure 3.12c). This is because the drones always try to offload all their computations, leading to high contention on the edge servers. Even though an offloading decision is taken only when this is beneficial, since the servers are jammed, the time savings are small for each such offloading. As a consequence, the aggregated time savings from all the computations are smaller than those of the offline

schedule. This, in turn, has an effect in the feasibility of the plan. Given that the aggregated energy consumption at the points of interest becomes larger than estimated by the offline algorithm, at some point the path becomes unsafe, leading to the insertion of an additional depot detour; it is these detours that produce the bottom outliers in the figures. In contract, the proposed approach respects the offline schedule and does not attempt to offload unless this is part of the offline plan. Moreover, the servers prioritize offloading requests based on the expected reduction of the drones' mission times, instead of using a blind FCFS policy.

Chapter 4

Learning Whether to Wait or Go in Data-Driven Missions

4.1 Introduction

In the previous chapters we focused our study in reducing the mission time of data-driven missions with multiple drones. First, we investigated the reduction of the flying time by efficiently planning the path of each drone, Chapter 2. Then, in Chapter 3, we explored how to leverage the processing capabilities of nearby edge servers to accelerate the data processing time, so that the hovering time of the drones is minimized.

In this chapter, we explore a different and largely orthogonal approach for reducing the time of such missions. Namely, instead of waiting at the current position until the computation finishes, we allow the drone to move to the next point of interest right after sensing. This way, time is gained if no action needs to be taken at the previous point of interest. However, if this decision is wrong, the drone must spend extra time to return to the previous point of interest to perform the required action. Our work focuses on the problem of taking the right decision at each point of interest thereby minimizing the time needed for the drone to execute the mission. To this end, we investigate a regression-based approach, which exploits the drone's past experience to take more informed decisions in future missions. We consider a single drone, but the approach can be applied horizontally on multiple drones; in fact, the proposed approach can be used to let drones learn from each other's experience.

The main contributions of the chapter are: (i) We provide a formal description of the system model and the above decision problem. (ii) We propose a heuristic that learns from the

experience gained from previous missions, using regression. (iii) We evaluate our approach via extensive simulation experiments, for a wide range of scenarios where the probability of action at each point of interest is stable or varies with time. (iv) We validate the accuracy of the simulator used to perform the experiments against the real flight behavior of a quadcopter drone and software-in-the-loop execution of the drone's autopilot.

The rest of the chapter is structured as follows. In Section 4.2 we give an overview of related work. Section 4.3 describes the system model and formulates the problem we tackle in this work. Section 4.4 provides the logic for controlling the drone so as to perform the mission at hand, independently of the method that is used to take the stay-or-go decision at each point of interest and Section 4.5 describes the learning heuristic. Section 4.6 presents the validation of the simulator used for our experiments, while Section 4.7 presents the evaluation of the heuristic and the machine-learning method for a wide range of scenarios.

4.2 Related Work

The problem we tackle in this chapter has similarities with other problems that have been investigated in the wider domain of autonomous vehicles and drones. Below we provide an overview and highlight the differences with our work (some of these works were also reviewed in the previous chapters but are also mentioned here for the sake of completeness).

Several research efforts focus on reducing the time it takes to perform heavyweight computations on autonomous vehicles, by offloading such computations to the cloud or nearby edge infrastructure. For instance, in [87] drones perform a 3D mapping in an unknown area, taking offloading decisions for the required computation tasks. In [76], multiple drones offload their tasks to the edge. The authors propose an offloading algorithm that predicts the channel quality based on historical data. In [68] the drone employs a runtime heuristic to decide whether it is worth offloading the computation to an edge server instead of performing it locally, so the time spent waiting for the result is minimized. This in turn reduces the mission time of the drone. Our work is complementary to these efforts, as it focuses on minimizing the mission time of the drone by taking smart decisions about whether to actually wait for such computations to complete before moving to the next waypoint.

[88] presents a drone-based system for detecting ground targets. One of the proposed strategies is to fly at high altitude covering a wider area and, if a target is detected, instruct

the drone to perform a close-up sensing. This is similar to the application pattern assumed in our work. However, the authors focus in the tradeoff between detection delay, coverage and detection quality, while we reduce the mission time by allowing the drone to move to the next point of interest before the results of the previous detection attempt become available.

A major component of total mission execution time in vehicle-related applications comes from traveling between the different points of interest. Thus, many works focus on the problem of path planning [89]. Also, a lot of work has been done in different vehicle routing problems, especially for electrical vehicles which have limited autonomy and must recharge or change batteries [90]. Other works use drones as a mobile edge server or base station, providing resources to nodes on the ground. For example, [65] uses a drone to serve mobile devices, optimizing the drone's trajectory based on the stochastic task arrival times. In [91], multiple drones are used to serve users, planning the drone trajectories so as to minimize the total energy consumption. [92] presents similar work for a UAV serving IoT devices. In our work, we take the (potentially optimized) drone path as input, and try to reduce the mission time by overlapping computation with flying.

The problem we study has some similarities with the vehicle routing problems (VRP) with stochastic aspects [36]. One such case is having a vehicle with limited supply capacity visiting customers in different locations to serve their stochastic demands which are not known in advance. If it turns out that the demand can not be satisfied, the vehicle must take some corrective action, e.g., return to a depot to resupply and return to the customer [93, 94, 95]. The cost of such potential detours is taken into account when planning the path of the vehicle. In our work, a similar corrective action is taken when the drone decides to go to the next waypoint without waiting for the computation to complete, but (after the computation ends) it turns out that further action is needed at the previous point of interest. The proposed decision approach takes into account the potential gains and penalty, weighted with the respective estimated probability, to minimize the mission time. Another key difference is that, in our case, the potential gain or penalty is affected not only by the probability of having to perform an additional action at the point of interest, but also (and quite substantially so) by the processing delay.

Less related to our application domain, a large body of literature has studied the problem of load estimation in cloud computing systems [96, 97, 98, 99]. The main issue at hand is to estimate the resources that will be required to serve a given job or application VM in order

to take more informed job scheduling decisions, e.g., regarding the node that will be used to run a new job, the migration of existing jobs on different nodes, and the packing of multiple jobs on fewer nodes (consolidation) so that other nodes can be turned-off to save energy. Similar to our work, this estimation is based on previous experience, typically using a form of regression over the loads recorded in previous scheduling periods. In addition, we study increased system dynamics scenarios, where the experience gained from previous missions becomes partly or even completely invalid and the decision making algorithm must discount or even completely ignore it.

4.3 System Model & Problem Formulation

This section presents a structured formulation of the problem we study in this chapter. We start by defining the mission and flight model. Then, we capture the relevant delays due to sensing, processing, action-taking and flight between the points of interest. Finally, we describe the decision problem at hand.

4.3.1 Mission model

We consider data-driven missions where the drone visits certain points of interest to perform some sensing, process the sensed data, and perhaps – depending on the outcome of the computation – perform an additional action on the spot.

We assume that the path of the drone is pre-computed using some planning algorithm (not the focus of this chapter). Let this be encoded as sequence of waypoints in the order they must be visited by the drone, $[wp_1, wp_2, ..., wp_N]$. The first and last waypoints wp_1, wp_N is the home location from where the drone takes-off and lands, while $wp_i, 2 \leq i \leq N-1$ correspond to the points of interest.

Let $senseT$ be the time that is needed for the drone to perform the required sensing at each point of interest, and $procT$ the time that is needed to process the sensed data. We assume that data processing generates a so-called detection event, only if a certain object or situation of interest is detected requiring further action from the drone. Note that such an event may not be generated, but this is not known before the computation is finished. Let e_i encode whether a detection event was generated for wp_i or not, taking values 1 and 0, respectively. Finally, let $actionT$ be the time that is needed for the drone to perform the additional action at the

point of interest if a detection event is generated ($e_i = 1$).

In this work, we assume homogeneous missions where the sensing, processing and action times are the same for every point of interest and can be reliably estimated offline or after a few test runs. Specifically regarding $procT$, we focus on completely autonomous drones that operate without the assistance of external infrastructure such as edge, cloud or 5G. As a consequence, all computations are performed locally on the drone's onboard computing platform where the respective computing time can be measured with great accuracy offline.

4.3.2 Flight model

We focus on polycopter drones, which can takeoff/ land vertically and can steadily hover above a specified location. Let $takeoffT$ be the time needed to perform a vertical take-off to the mission altitude, and $landT$ be the time needed for a vertical landing. Also, let $flightT(wp_i, wp_{i+1})$ be the time it takes for the drone to fly from wp_i to wp_{i+1} while maintaining the mission altitude. To focus on the essence, we assume an obstacle-free mission area, where the drone can always travel between waypoints in a straight line. Therefore the flight time is a direct function of the distance between two waypoints. Note that it is quite straightforward to generalize the model to capture the case where the drone must follow a different trajectory to avoid an obstacle between two waypoints, but this does not change the core problem at hand. When the drone reaches a waypoint, it performs the sensing task while hovering at that position. Then, it either waits for the computation to finish while still hovering over the point of interest, or starts moving to the next point of interest immediately after sensing.

Note that $flightT(wp_i, wp_{i+1})$ also depends on the target cruising speed of the drone, which, in turn, affects the acceleration when moving away from a waypoint and the deceleration when approaching the next one. Similarly, $takeoffT$ and $landT$ depend on the cruising altitude and desired vertical takeoff/landing speed. In this work, we consider such parameters to be fixed features of the drone and/or mission at hand. However, it is possible to generalize the system model so that that such aspects become open parameters, but again this does not change the core problem at hand. Finally, we assume that the flight direction between two waypoints does not affect the flight time, so $flightT(wp_i, wp_{i+1}) = flightT(wp_{i+1}, wp_i)$.

4.3.3 Decision, time gain and time penalty

After sensing at waypoint wp_i, the drone must decide whether it will keep hovering at that location waiting for the completion of the data processing computation, or move on to the next waypoint wp_{i+1}. Let this decision be encoded as d_i, taking values 0 and 1, respectively.

We take as a "baseline" the decision to wait at wp_i for the computation to finish before moving to the next waypoint ($d_i = 0$), which is the default behavior under conventional operation. Then, the drone will stay at the given point of interest for $senseT + procT$.

We now focus on the case where the drone decides to move from wp_i to wp_{i+1} as soon as sensing is completed ($d_i = 1$). If no detection event is generated as a result of data processing ($e_i = 0$), it will save $procT$ vs having waited at wp_i for the computation to finish. If, however, a detection event is generated ($e_i = 1$), the drone will spend additional time vs having simply waited at wp_i. This is because it must stop moving towards wp_{i+1} and start moving backwards to approach wp_i so that it can perform the required action at that location.

Let the time penalty for this be $returnT(wp_i, wp_{i+1}, procT)$. Note that $procT$ is a decisive parameter of the return penalty as it determines the distance the drone has already traveled by the time it discovers that it needs to return to the previous waypoint. Like the time that is needed for the drone to move from one waypoint to the next one, $flightT(wp_i, wp_{i+1})$, the return penalty also depends on the target cruising speed and the corresponding acceleration/deceleration when starting/stopping the drone. As mentioned above, in this study, we consider these parameters fixed properties of the drone/mission.

Note that if the drone chooses to go, it may arrive to the next waypoint wp_{i+1} before the computation for the previous waypoint wp_i is completed. This will happen in the special case where $procT \geq flightT(wp_i, wp_{i+1})$. To keep our model simple, we assume that in this case the drone waits at wp_{i+1} until the previous computation finishes. If no event is detected, it continues with the sensing and processing tasks for wp_{i+1} as usual. Else, the drone goes back to wp_i to perform the required action. Note that, in this special case, $returnT(wp_i, wp_{i+1}, procT) = flightT(wp_{i+1}, wp_i)$.

4.3.4 Mission time

Based on the above, the time it takes for the drone to perform a given mission, including the takeoff and landing times, can be calculated using Equation 4.1 as shown below:

$$missionT = takeoffT + flightT(wp_1, wp_2) + \sum_{i=2}^{N-1}(visitT_i) + landT \tag{4.1}$$

$$\begin{aligned} visitT_i = & senseT + procT + flightT(wp_i, wp_{i+1}) \\ & + e_i \times d_i \times returnT(wp_i, wp_{i+1}, procT) \\ & - (1 - e_i) \times d_i \times procT \\ & + e_i \times actionT \end{aligned} \tag{4.2}$$

where $visitT_i$ as given in Equation 4.2 is the total time spent by the drone to fully complete the visit of wp_i. This includes the "baseline" time (first line), adding the penalty when the drone wrongfully decides to move to the next waypoint before the computation finishes and must return to the previous waypoint (second line), and, conversely, subtracting the gain when the drone correctly decides to move on (third line). It also includes the time for performing a follow-up action if a detection event is generated (fourth line).

4.3.5 Problem

The goal is for the drone to take decisions d_i at each point of interest wp_i so as to minimize $visitT$ and consequently also $missionT$. However, the generation of the detection events e_i is unknown to the drone or the mission planner. It is only discovered at runtime, when the drone completes processing the data collected at wp_i. More specifically, we focus on the general case where e_i is not deterministic but is governed by an underlying probability p_{wp_i}. In other words, we assume that the probability of the drone needing to take further action may depend on the specific point of interest/location.

The problem is that the underlying detection probability p_{wp_i} is unknown to the drone. Therefore, in order to minimize the mission time, the drone must be able to *learn* what is the best decision to take at each waypoint, based on the experience it accumulates over time. Furthermore, the drone must be able to *adapt* its decisions in case the system is dynamic and the event detection probability at the points of interest changes with the passage of time.

In the following, we describe how the drone can be programmed to execute the mission at hand, independently of the method used to take the stay-or-go decision at each point of

interest. Then, we present an approach to tackle the decision problem, that learns by applying data analysis (regression) on the data gathered from previous missions.

4.4 Mission Execution

At each point of interest, the drone must take a decision, to stay or go. Depending on the actual event generation, this decision may turn out to be correct or wrong. Nevertheless, the drone can be programmed to execute the mission in a generic way, independently of the method that is used to take these decisions. It suffices to handle any possible outcome of each decision as needed.

Algorithm 11 Drone control logic.

```
DecisionMethod.MissionBegin(wp[])
Autopilot.Arm&TakeOff()
Autopilot.GoTo(wp_2)
for i from 2 to len(wp) − 1 do
    Autopilot.WaitToArrive()
    data ← Sense()
    StartComputation(data)
    d ← DecisionMethod.Decide(wp_i, wp_{i+1}, procT)
    if d = 1 then                          ▷ immediately move to the next waypoint
        Autopilot.GoTo(wp_{i+1})
    end if
    e ← GetComputationResult()
    DecisionMethod.Feedback(wp_i, wp_{i+1}, d, e)
    if e = 1 then
        if d = 1 then                      ▷ return back to perform action
            Autopilot.GoTo(wp_i)
            Autopilot.WaitToArrive()
        end if
        PerformAction()
    end if
    if d = 0 ∨ e = 1 then                  ▷ now move to the next waypoint
        Autopilot.GoTo(wp_{i+1})
    end if
end for
Autopilot.WaitToArrive()
Autopilot.Land&Disarm()
DecisionMethod.MissionEnd
```

Algorithm 11 gives a high-level description of the mission logic used to control the drone by sending corresponding commands to the autopilot. In a nutshell, the drone takes-off and

starts visiting the points of interest according to the specified mission plan. When it arrives at a point of interest, it performs the sensing task and starts the computation to process the data collected. Then, the drone decides whether to stay waiting for the computation to finish, or move to the next point. In the first case, the drone just waits hovering above the point of interest. If the computation generates a detection event then the drone performs the necessary action, before moving on. In the second case, the autopilot is instructed to go to the next waypoint and the drone waits until it receives the results of the computation. If no event is generated, the mission proceeds as usual, else the autopilot is instructed to go back to the previous waypoint where the drone performs the required action. This procedure is repeated until all points of interest have been successfully visited. After visiting the last point of interest, the autopilot is instructed to return and land the drone.

The method used to take the stay-or-go decision at each point of interest is abstracted in the form of a software component (DecisionMethod) that is invoked by the mission program through a structured API. It is important to stress the fact that the decision component can have rich and potentially persistent internal state, which may be preserved across mission executions to encode experience from previous mission executions so that this knowledge can be used to take decisions in the next mission. This state is initialized (but not necessarily reset) via MissionBegin() before the mission starts and it is updated via MissionEnd() after the mission ends. The decision at each point of interest is taken via Decide() while the feedback regarding the correctness of this decision is communicated back to the component via Feedback(). This abstraction simplifies experimentation with different decision methods without changing the mission program.

4.5 Heuristic Method

In this section, we present a heuristic that tries to learn the underlying event detection probability by gathering and analyzing experience that was gained from previous missions.

4.5.1 Basic learning approach

The heuristic works based on data collected from previous missions. We refer to this historical data as the *experience memory*. Initially, the experience memory is empty and is incrementally filled with each mission.

During the execution of a mission, DECISIONMETHOD.FEEDBACK() is used to record whether a detection event was generated for each waypoint at the time (day) when the drone performed the required sensing task. This information is stored in the form of separate entries $\langle wp_i, e_i \rangle$, where wp_i is the point of interest and e_i denotes whether the data processing led to a detection event that requires further action from the drone. The entries are kept in a list, sorted according to the time they were generated (the most recent entry is at the head of the list and the oldest one is at the tail).

Before starting a new mission. via DECISIONMETHOD.MISSIONBEGIN() a regression algorithm is run over the experience memory to estimate the event generation probabilities p_{wp_i} for the upcoming mission. The resulting probabilities are then used to take the stay-or-go decisions during the mission via DECISIONMETHOD.DECIDE(). After the mission has been completed, the experience memory can be post-processed via DECISIONMETHOD.MISSIONEND(). This is optional, depending on the variant of the heuristic (see next), so the respective implementation can be empty.

Our design is modular, making it possible to employ different regression algorithms in a plug-and-play manner. In the evaluation that is presented in Section 4.7, we explore three different configurations for the regression algorithm: (i) linear regression [100], (ii) decision tree regression [101], and (iii) bayesian ridge regression [102]. Notably, no explicit assumptions are made about the potentially spatial dependence of p_{wp_i}. We simply pass to the regression algorithms(s) all the data from the experience memory.

4.5.2 Experience memory size & reset

In the general case, it is not realistic to assume infinite experience memory. Also, if the detection event probability changes with time, past experience becomes less useful or may even lead to completely wrong decisions. For this reason we only keep the H most recent entries for each wp_i. We refer to H as the size of the experience memory. In the evaluation, we experiment with different values for H. We also consider the case of infinite experience memory ($H = inf$).

Note, however, that even relatively small experience memory sizes will not work well if p_{wp_i} changes significantly between missions. The reason is that, in this case, even recent experience can be very misleading. To handle such scenarios, we introduce a reset mechanism that clears the experience memory, by dropping all entries except the ones corresponding to

the recent mission, whenever an anomaly is detected. As a result, the drone will start the next mission practically without any past experience, except that of the previous mission, and will basically learn from scratch. We experiment with three anomaly detection methods, described in the following.

The first anomaly detection method works as follows. During mission execution, we see whether a decision was correct or wrong via DecisionMethod.Feedback(), and in the latter case we record the time penalty due to the wrong decision. After the mission ends, DecisionMethod.MissionEnd() is used to aggregate the individual time penalties to a total penalty for the entire mission. An anomaly is signalled if the penalty for the recent mission becomes greater than the maximum penalties that were experienced over the first H missions. The rationale is that if the drone, after having gone through several learning periods, performs worse than it did during the initial learning period, this is a strong indication that recent experience is no longer valid. We refer to this as the initial static threshold (IST) method.

As another option for anomaly detection, we investigate the interquartile range (IQR) method. Unlike the IST method, this works on the penalties that were experienced during the last H missions, again recorded during mission execution via DecisionMethod.Feedback() and aggregated per mission via DecisionMethod.MissionEnd(). Given these penalties, we calculate the 25^{th} and 75^{th} percentiles, Q1 and Q3 respectively, and the interquartile range $IQR = Q3 - Q1$. Typically, a data point is considered outlier if it is greater than $Q3 + 1.5 \times IQR$ or smaller than $Q1 - 1.5 \times IQR$. However, in our case, it is not desirable to reset the experience memory if the drone takes good or even optimal decisions which result in a very low mission penalty. For this reason we consider the last mission as an anomaly only if the respective penalty is greater than $Q3 + 1.5 \times IQR$, i.e., the drone has experienced a significant penalty compared to previous missions.

As a third anomaly detection method, we use the Local Outlier Factor [103] (LFO), which is an unsupervised outlier detection method. Like in IQR, we apply this on the penalties experienced in the last H missions. For each data point (mission penalty) an anomaly score is calculated depending on how much isolated the point is from its k-nearest neighbors. The more isolated a point is, the more likely it is to be an outlier. More precisely, the distances between a given point and each of its k-nearest neighbors are used to calculate the so-called local density of that point. If this is significantly lower than the local density of its neighbors, then that point is considered an outlier. In our case, when we calculate the anomaly score of

the last mission, we consider the $(H-2)$ nearest neighbors from the H last missions (leaving one point out, to avoid an outlier point to be used to calculate the distances of the non-outlier points). If the penalty of the last mission is marked as an outlier *and* it is greater than all other H penalties, this is considered an anomaly.

Note that IST depends on the penalties the drone experiences during its initial learning phase. One disadvantage of this approach is that, if the drone, by chance, performs well in the first missions, this may lead to unnecessary memory resets in the future, especially if the underlying probabilities change with time. On the other hand, the IQR and LFO methods that depend on the experience of the last H missions, may lead to an unnecessary reset if the drone performs poorly out of bad luck, although its decisions are well-informed (based on estimations that are close to the underlying event detection probabilities). In any case, if the probabilities change marginally, it could be beneficial to keep the experience gained from previous missions rather than learn from scratch. For the above reasons, we consider another, more strict, anomaly detection variant, which flags the recent mission penalty as an anomaly only if this is considered so by both LOF and IST. We refer to this as the *LOF+IST* method.

4.5.3 Decision making

The decision to stay or go, returned by DecisionMethod.Decide(), is taken based on the calculated event detection probability p_{wp_i} so as to minimize the expected penalty due to a wrong decision.

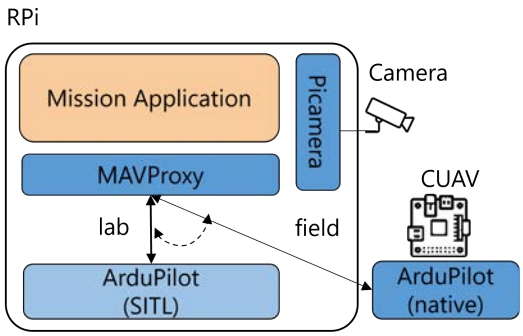
More specifically, this is done using Equation 4.3 below

$$d_i \leftarrow \arg\min_{c=[0,1]} EP(c, wp_i) \tag{4.3}$$

$$EP(c, wp_i) = \begin{cases} (1 - p_{wp_i}) \times procT, & c = 0 \text{ (wrong decision to stay)} \\ p_{wp_i} \times returnT(wp_i, wp_{i+1}, procT), & c = 1 \text{ (wrong decision to go)} \end{cases} \tag{4.4}$$

where $EP(c, wp_i)$ as per Equation 4.4 is the expected time penalty at point of interest wp_i if the drone takes decision c, which is calculated as the product of the penalty and the probability of the decision being wrong. In turn, Equation 4.3 picks the choice that minimizes the expected penalty.

(a) Custom drone.



(b) Autopilot in field vs SITL in the lab.

Figure 4.1: Drone in the field vs lab setup using the SITL configuration of the autopilot.

It is important to stress that the choice c that minimizes $EP(c, wp_i)$ depends both on the (estimated) probability of event detection p_{wp_i} and the penalty of a wrong decision. As a consequence, the decision d_i that is taken by the heuristic at wp_i may not strictly obey the respective event detection probability p_{wp_i}. For example, if $returnT(wp_i, wp_{i+1}) > procT$ then the best decision at wp_i could be to stay ($d_i = 0$) even though $p_{wp_i} < 0.5$, as the potential penalty of the drone having to return to wp_i may outweigh the fact that it is less likely for a detection event to be raised at wp_i than not.

4.6 Validation of High-level Simulator

In this section, we explain how we have validated the high-level simulator that is used for the extensive evaluation of the above decision method. As a first step, we run a mission in the field using a real drone, and compare the observed flight behavior vs running the same mission with the autopilot in software-in-the-loop (SITL) mode. Then, we compare the results we get for a more complex mission scenario when using the SITL autopilot vs the high-level simulator.

Our tests show that the differences are small hence the high-level simulator can serve as a practical evaluation tool, without having to do real field tests. Note that the latter is not only time-consuming but more importantly makes it practically impossible to reproduce the same conditions in each test, which is needed to be able to run the same scenario multiple times. In contrast, using the high-level simulator one can make a fair and reproducible comparison between the different decision methods.

4.6.1 Drone setup

For the field experiments, we use a custom quadcopter drone, shown in Figure 4.1a. This runs the ArduPilot autopilot [86] on a CUAV Nano v5 board [104]. The drone is also equipped with a Raspberry Pi (RPi) with a camera, serving as an indicative on-board sensing and computing platform. The RPi is connected to the autopilot board via serial and hosts the application code, which runs the desired mission by retrieving the status and sending commands to the autopilot using the MAVLink protocol [105] through the MavProxy library [106].

As an indicative processing task, we let the drone run an object detection algorithm on the pictures taken by the camera on top of the points of interest. The RPi takes about 1 second to capture an image and from 8 to 12 seconds (on average 10 seconds) to perform the computation for object detection. This is done using yolov8x [107] for a 1080p image downscaled to 480p and 640p, respectively. We have chosen at least 480p as the image input size and the slowest but most accurate yolov8x model to allow objects to be detected with 0.95 confidence.

4.6.2 Validation of SITL vs real flight behavior

For practical reasons, the real-world flight experiments are performed in an empty field where the drone moves back-and-forth between two waypoints (WP1 and WP2) 50m apart, representing points of interest. At WP1, the drone performs the required sensing and immediately starts moving towards WP2. We assume that processing the data collected at WP1 always raises a detection event, so the drone stops moving towards WP2 and returns to WP1. The opposite case is tested when the drone visits WP2, where it always waits for the computation to finish before moving to WP1. All movements between the two waypoints are performed in a straight line at a target cruising speed of 4m/s. Note that, despite its simplicity, this test scenario fully captures the basic flight behavior of the drone that is relevant for our experiments.

Note that the results of the computation do not affect the drone's decisions, which are hardcoded in the mission program as described above. For this reason, in the above experiments, instead of actually performing the computation, the RPi sleeps for a predetermined amount of time. This allows us to run tests for different values of $procT$ equal to 8, 10 and 12 seconds, to reproduce scenarios where the drone performs a more lightweight or heavyweight computation, or features a more or less powerful on-board computing platform, respectively.

For each scenario, we let the drone repeat these waypoint visits for 10 consecutive times, and record the total mission delay as well as the time it takes for the drone to stop moving towards WP2 and return to WP1. We repeat these experiments with the autopilot running in the SITL configuration [85] coupled with a physics model that simulates the dynamics of the aerial vehicle. The respective software configuration is shown in Figure 4.1b. Note that the mission code remains exactly the same as in the field tests except that it interacts with the SITL autopilot configuration instead of the autopilot board. Overall, the results of the SITL experiments are very close to those of the field experiments, with an average deviation of the mission time about 10%. This confirms that the SITL configuration can reproduce the drone behavior with sufficient accuracy.

4.6.3 Validation of high-level simulator vs SITL

As an alternative to the SITL-based mission execution, we have developed a lightweight high-level simulator that captures all the delays of the drone's movements that are relevant for our study, based on the SITL measurements, without using a real autopilot or an underlying physics model.

To validate the high-level simulator vs the SITL-based mission execution, we use a more complex mission. More specifically, the drone visits 8 waypoints, in half of which the drone decides to wait for the computation to finish while for the other half it decides to go to the next waypoint. At each waypoint, we manually set event generation so that half of the drone's decisions are correct and the other half are wrong. Thus, there are 2 cases where the drone wrongfully decides to move to the next waypoint before the computation finishes and must return back to the previous waypoint. Also, in 2 cases the drone takes the wrong decision to wait for the computation even though this does not generate a detection event.

We run the same mission using the SITL configuration and the high-level simulator, with the processing time $procT$ set to 10 seconds. The target cruising speed of the drone is set to 4m/s as in the field tests. We record all relevant events, including time to take-off and land, flight time between waypoints, time spent at each waypoint and the time penalty when deciding to go but then need to return back to the previous waypoint. Figure 4.2 shows the event timeline for each of the two executions. The black events correspond to the execution of the mission using the SITL configuration, while the red events correspond to the mission execution using our high-level simulator. The time difference for the same events is in the

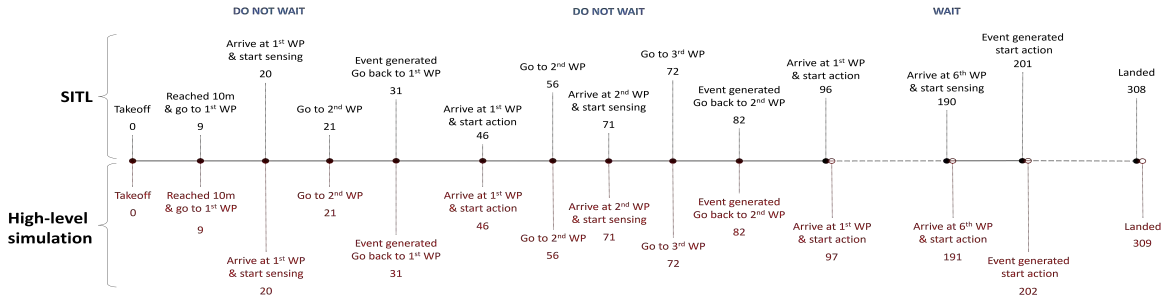


Figure 4.2: Timeline of a complex mission execution scenario using the autopilot in SITL mode vs the high-level simulator.

order of a few milliseconds, leading to an aggregate deviation of merely a couple of seconds for the entire mission.

We have repeated the experiment with several variations for $procT$ set to 8 and 12 seconds as well as for a different number of waypoints and event detection scenarios, confirming that the high-level simulator faithfully reproduces the results of the SITL-based mission execution. However, the latter is far more time-consuming (it takes the same time as if the drone were actually flying in the real world), whereas the high-level simulator takes just a few seconds. Therefore, we conduct our extensive evaluation using the high-level simulator. As an extra sanity check, we have run selected cases from those experiments via SITL, which produced practically identical results.

4.7 Evaluation

In this section, we present a wide range of experiments through which we observe and discuss the results achieved by the proposed decision method. Our evaluation captures different scenarios where the probability of event detection remains stable or varies between missions.

We start by describing the basic mission scenario and how the event detection probabilities vary in space and with the passage of time (between missions). We also introduce some simple decision policies serving as benchmarks. Then, we proceed with the evaluation. First, we study different configurations of the heuristic to determine a variant with overall good performance. Then, we compare this configuration of the heuristic with the benchmark policies.

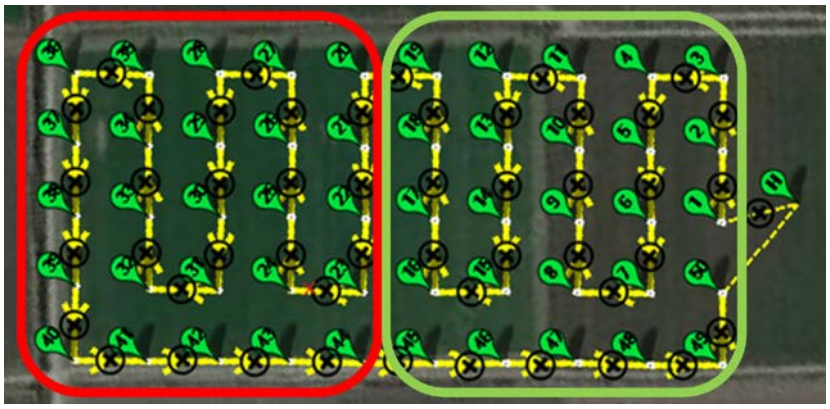


Figure 4.3: Mission area with two regions (red and green rectangles) and path plan (yellow line).

4.7.1 Mission plan & event detection probability scenarios

For our evaluation we use a 450m x 200m area of interest, consisting of 50 waypoints spaced 50m apart in grid-like manner, as shown in Figure 4.3. In all experiments, the drone starts from a nearby home location (H), visits all waypoints following the same pre-specified path plan, and returns to home. The drone's path is indicated by the yellow line in the figure. We let the drone execute the same mission over a period of several days (depending on the scenario) and study how well the proposed method guide the drone to take good decisions.

To introduce spatial heterogeneity in terms of event detection, we separate the mission area in two regions denoted by the colored rectangles in Figure 4.3, with a different event detection probability; however, all points of interest in the same region have the same event detection probability. We experiment with various cases regarding the event detection probabilities of these regions, listed in Table 4.1, from the case where the two regions have significantly different event detection probability, to the case where these differences are rather small. Also, there are several cases where the regional event detection probability is close or equal to 0.5, which makes probability estimation based on the observed detection events harder compared to the cases where this is closer to 0.0 or 1.0.

In addition, we investigate three different cases regarding the dynamics of the regional event detection probabilities and how these may change with time, summarized in Table 4.2. More specifically, we study a so-called stable world where the regional probabilities remain fixed throughout all days of operation, as well as two cases where these probabilities change with time. On the one hand, we study a gradually changing world where the event detection probabilities of the two regions change in a smooth way, by 0.05 each day, until they are

Table 4.1: Regional probabilities.

Label	Green	Red
0.2/0.8	0.2	0.8
0.4/0.6	0.4	0.6
0.1/0.5	0.1	0.5

Table 4.2: Change of regional probabilities in time.

Label	Description
Stable	Regional probabilities remain constant over time.
Gradual change	Regional probabilities change gradually until they are reversed.
Abrupt change	Regional probabilities are reversed abruptly without any smooth transition period.

completely reversed, i.e., the points of interest in the red region end-up having the initial event detection probability of those in the green region and vice versa. On the other hand, we study a so-called abrupt world change where the regional probabilities are reversed in a single day.

The combination of the above cases results in a wide range of scenarios with significantly different features. This is important to evaluate the robustness and general applicability of the decision method we investigate in this work.

4.7.2 Benchmarks and reference for performance comparison

In our evaluation, we compare the performance of the proposed decision method with the results that are achieved by another four different decision policies, briefly described below.

- **Wait:** The drone acts in a blindly conservative way and always waits for the computation to complete before moving to the next point of interest.

- **Go:** The drone acts in a blindly optimistic way and always moves to the next point of interest right after sensing completes.

- **Random:** The decision whether to wait or to go at each point of interest is taken by flipping a coin.

- **Knowledgeable:** This works like the proposed heuristic, taking decisions based on Equation 4.3 and Equation 4.4, by weighing the probability of a wait/go choice being wrong at each point of interest with the respective penalty. However, unlike the heuristic, this policy has *perfect* knowledge of the true underlying event detection probability at each point of interest.

Given that the knowledgeable policy takes decisions based on the true event detection probabilities, we use it as a reference for all other methods and policies. In other words, the mission times that are obtained when using other methods/policies are presented relative to the mission time that is achieved by the knowledgeable policy. Since other methods/policies will generally take worse decisions, the respective mission times are expected to be longer than or in the best case equal to that of the knowledgeable policy. Note, however, that since event generation is random, even the knowledgeable policy can take wrong decisions. More specifically, this will happen if an event detection does (not) occur at a given point of interest/time against the true underlying probability. In this case, another method/policy may take a better (but merely lucky) decision. Still, this is circumstantial and highly unlikely to lead to systematically better results.

All the above policies can be integrated into the mission execution logic (Algorithm 11) in a straightforward way, as different implementations of the DecisionMethod component. The stay and go policies are trivial and completely stateless. The random policy merely initializes and uses a random number generator to take 50-50 decisions, while the knowledgeable policy stores and uses the true event detection probabilities for each scenario.

4.7.3 Scenario traces

To make a fair comparison between our proposed method and the benchmark policies used to take the stay-or-go decision at each point of interest, it must be possible to run exactly the same tests multiple times. Since event generation is probabilistic, extra care is needed to ensure that in each test the drone will observe the same presence or absence of a detection event at each point of interest.

To this end, the detection events for each test scenario (different cases in Table 4.1 and Table 4.2) are generated offline, based on the respective regional probabilities. This information is recorded in a trace file, which is then replayed during mission execution. More specifically, each trace includes the presence/absence of a detection event at each point of interest and each day (consecutive mission executions). Note that the number of days depends on the scenario, namely changing world scenarios extend over more days than for the stable world, to observe how the heuristic adapts to changes and converge to a stable behavior once the change is over.

We generate 20 random traces for each test scenario. Note that each such trace corre-

sponds to an independent experiment where the event detection probabilities are learned from scratch. The results presented in the sequel are the averages over the respective mission executions. Unless stated otherwise, $procT$ is set to 10 seconds, which is the measured average time it takes to run the object detection algorithm on our drone (see Section 4.6.1).

4.7.4 Different regression algorithms

As a first step of our evaluation, we study the performance of the heuristic using the three different regression algorithms for estimating the underlying event detection probabilities, discussed in Section 4.5.1. We focus on the stable world for the three regional event detection probability scenarios in Table 4.1. The size of the experience memory is set to be infinite ($H = inf$), so the heuristic does not forget any past experience and continuously accumulates knowledge that is fed into the regression algorithms to update the estimated event detection probabilities after each mission (day). The duration of the experiments is 30 days to observe convergence over a longer period of time. Note, that in all of the experiments conducted, when the drone uses the heuristic to decide and does not have any previous experience (1^{st} day), it chooses to wait.

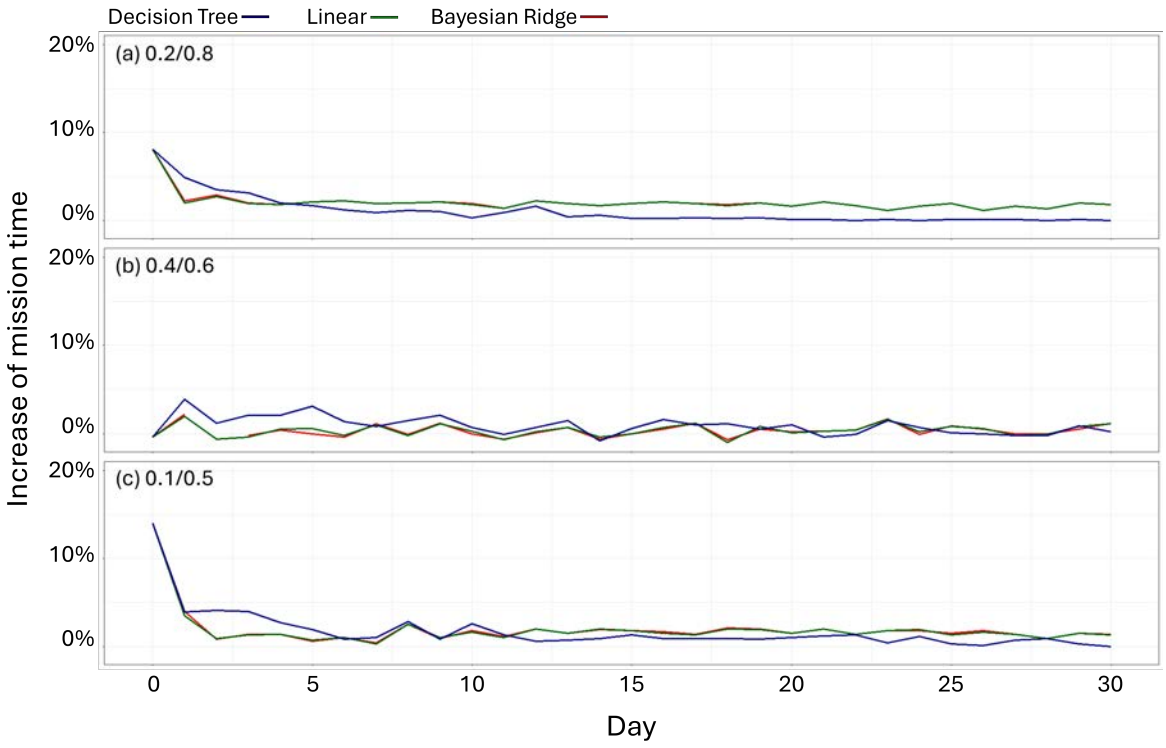


Figure 4.4: Heuristic with different regression algorithms in the stable world.

The results are shown in Figure 4.4. Each line plots the relative increase of the mission

time achieved vs the knowledgeable policy (lower is better). As can be seen, all regression algorithms lead to a steady behavior in the $0.2/0.8$ scenario, while being more unstable in $0.1/0.5$ and even more so in the $0.4/0.6$ scenario. This is expected given that the closer a regional probability is to 0.5, the more likely it is for even small inaccuracies in the estimation of the probabilities to lead to wrong decisions (different than those taken by the knowledgeable policy).

The linear and bayesian ridge regressions perform better in the first days of the experiments. In other words these algorithms make a good probability estimation with few data. However, the decision tree regression outperforms the other two after some experience is gained. More precisely, the average relative increase in the mission time vs the knowledgeable policy that is achieved after day 10 and before day 20, is 0.7%, 1.2% and 1.2% for the decision tree, the linear and the bayesian ridge regressions. After day 20 until day 30, the average performance of the heuristic for each of the three regression algorithms becomes 0.3%, 1.2% and 1.2% respectively.

Overall, the decision tree regression leads to better performance after a relatively short learning period. For this reason, in the rest of the evaluation, we run the heuristic with this regression algorithm.

4.7.5 Different sizes of the experience memory in a stable world

In the next set of experiments, we keep the decision tree regression approach and study how the heuristic performs in a stable world for different sizes of the experience memory H=8, 12, 16, 20. We also run experiments for $H = inf$, where the drone never forgets any of the past experience. The results are presented in Figure 4.5 for the three regional event detection probability scenarios in Table 4.1. Again, the lines plot the relative increase of the mission time vs the mission time achieved by the knowledgeable policy.

We observe that most memory-limited configurations perform very close to the one with the infinite experience memory, converging to the performance achieved by the knowledgeable policy by day 12. The results achieved for H=8 are slightly but quite consistently worse than the rest of the configurations, indicating that this size is insufficient. More precisely, after day 20 where all memory limited configurations have filled their experience memory to its maximum capacity, the relative increase of the mission time for H=8, 12, 16, 20 is 1.2%, 0.8%, 0.7% and 0.6%, respectively, averaged over all mission scenarios.

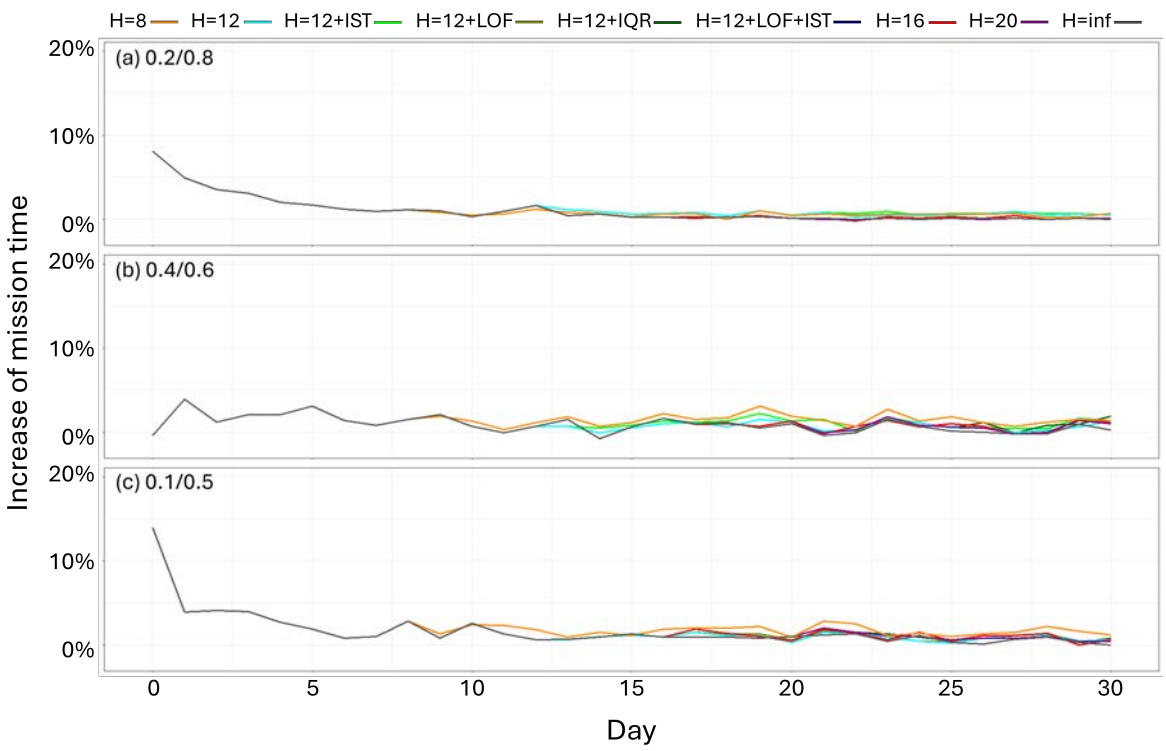


Figure 4.5: Heuristic with different sizes of the experience memory in the stable world.

Since $H \geq 16$ does not improve the results significantly, we keep the H=12 configuration and perform experiments for each of the memory reset options (IST, IQR, LOF, LOF+IST). The results are also shown in Figure 4.5. An interesting observation is that IST is less stable than all other H=12 configurations. This is because IST identifies anomalies and resets the experience memory even though the underlying event detection probability does not change. More specifically, IST has a false positive rate of 0.3%, 2.8% and 0%, in the $0.2/0.8$, $0.4/0.6$ and $0.1/0.5$ scenarios, respectively. The high false positive rate in the $0.4/0.6$ scenario is because both regional probabilities are very close to 0.5, thus even fully informed choices have a high probability of turning out to be wrong vs the actual event generation at each point of interest during mission execution. Also note that, in this particular case, performance during the first missions is quite good, which makes IST more sensitive to falsely resetting the experience memory in the future. In comparison, the false anomaly detection rate of IQR is 0%, 0.6% and 0.6% in the $0.2/0.8$, $0.4/0.6$ and $0.1/0.5$ scenarios, while LOF has 0%, 0% and 0.3% respectively. Notably, LOF+IST does not perform any false resets as it implements a more strict anomaly detection.

The average (over all scenarios) relative increase in mission time for H=12 with the IST, IQR, LOF and LOF+IST reset logic after day 12 (where the experience memory has been

filled) is 0.9%, 0.8%, 0.8% and 0.8%, respectively. Note that resetting the experience memory does not improve performance vs the simple H=12 configuration. On the contrary, in some cases, it leads to slightly worse results. This is expected given that the world is stable and the underlying event detection probabilities at the points of interest remain constant. However, as will be shown in the sequel, resetting the experience memory becomes important when the world changes.

4.7.6 Different sizes of the experience memory in a changing world

Next, we explore the performance of the heuristic for a world that changes according to the last two cases of Table 4.2. First, we study the case where the world changes gradually, and then the case where the world changes abruptly. In each case, we perform experiments for two scenarios, where the world starts with the regional event detection probabilities $0.2/0.8$ and $0.1/0.5$ which are subsequently revered (in a gradual or abrupt way) to $0.8/0.2$ and $0.5/0.1$, respectively.

Gradually changing world

In this set of experiments we explore the performance of the heuristic for a world that changes gradually. The world starts changing on day 21, after a stable period of 20 days, and the experiment goes on for another 40 days to observe the adaptation after the change is over. The results are shown in Figure 4.6, where the start and end of the world change is indicated by the vertical dashed lines. Note that the period of change in Figure 4.6(a) is longer than in Figure 4.6(b). This is because the difference between the regional probabilities in the $0.2/0.8 \rightarrow 0.8/0.2$ scenario is larger than in the $0.1/0.5 \rightarrow 0.5/0.1$ scenario (0.6 vs 0.4), while the step of change is the same (0.05) in both scenarios. Thus, the period of change in the first scenario is $1.5x$ that of the second scenario. As in the previous figure, the lines indicate the relative increase of the mission time achieved by each configuration of the heuristic vs the knowledgeable policy.

Looking at the results, we see that all configurations of the heuristic have the same adaptation profile, namely the performance increasingly deteriorates during the period of change. The day on which performance starts getting worse depends on when the drone should start taking different decisions in some points of interest, given the changed/new probabilities. This happens after the 5^{th} day of change in the $0.2/0.8 \rightarrow 0.8/0.2$ but already at the and 2^{nd}

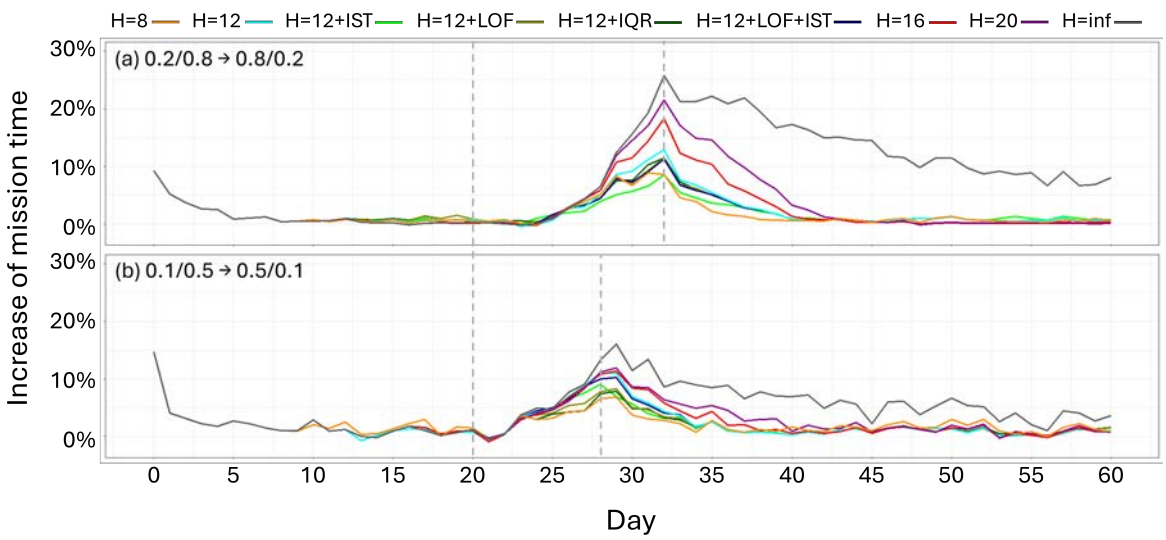


Figure 4.6: Heuristic with different sizes of the experience memory in the gradually changing world.

day in the $0.1/0.5 \rightarrow 0.5/0.1$ scenario, because changes around 0.5 lead sooner to the point where a different decision should be taken at the respective points of interest. Notably, in both scenarios, performance keeps getting worse until the world stops changing. This is because the heuristic does not manage to adapt the internally estimated probabilities as fast as the world is changing – recall that the regional probabilities change, even if slightly, each day. After the period of change is over, performance improves as the heuristic gradually learns the respective stable underlying event detection probabilities. The bad performance during the period of change is more significant for the configurations with larger memory sizes as they keep more outdated information. For the same reason, when the change is over, the configurations with the smaller experience memory sizes converge faster to the knowledgeable policy. Infinite memory size is clearly problematic in both scenarios where the world changes.

All H=12 configurations which reset the experience memory detect the changes and adapt faster than the simple H=12 configuration without any reset logic. Note that in the $0.2/0.8 \rightarrow 0.8/0.2$ scenario, IST performs slightly better than the other reset methods during the period of change. The reason is that in this case performance during the change tends to get worse than the performance of day 0 (the worst performance in the first 12 days/missions). As a result, IST resets the experience memory sooner and in more cases than the other reset methods, which experience gradually increasing penalties in the 12 most recent missions hence do not detect the anomaly as soon, or at all, as IST. However, in the $0.1/0.5 \rightarrow 0.5/0.1$ scenarios, IQR and LOF detect the anomaly better than IST which does not reset the experi-

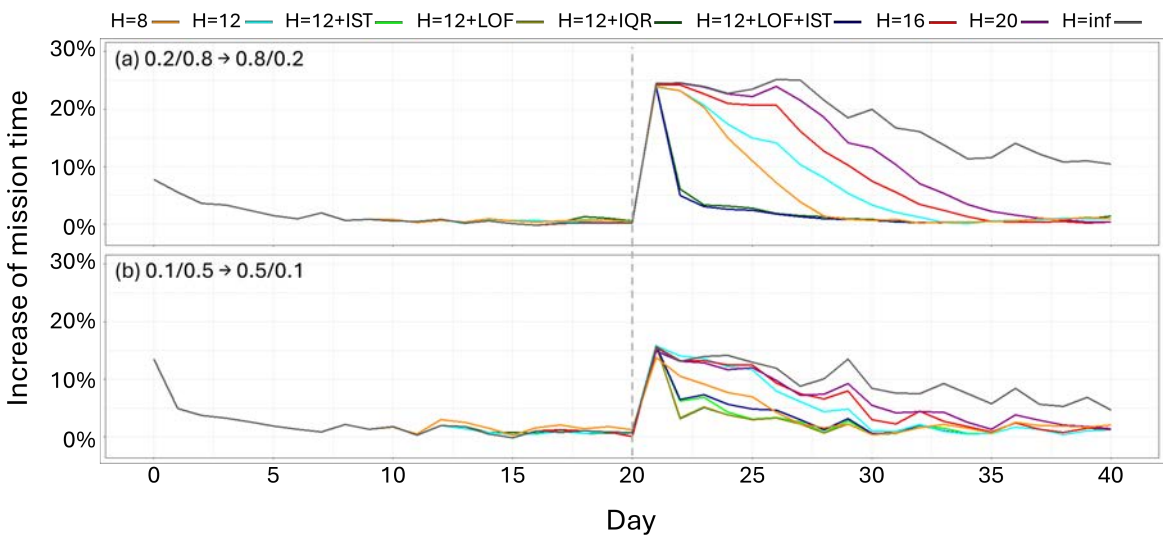


Figure 4.7: Heuristic with different sizes of the experience memory in the abruptly changing world.

ence memory as aggressively since the penalty of day 0 is larger compared to the first missions that take place during the period of change. Finally, LOF+IST performs only slightly better than the simple H=12 variant without the reset as it does not detect an anomaly in most cases.

Abruptly changing world

Next we focus on the case that the world changes abruptly. Like in the previous scenarios, the world changes after day 20. However, the change does not take place in a gradual way over a longer period, but happens abruptly on day 21. The experiments continue for another 20 days to observe the adaptation behavior of the different configurations of the heuristic.

The results are shown in Figure 4.7. We observe that on the day of change (marked by the vertical dashed line), the performance becomes worse than on day 0, for all configurations. This is expected since most of the decisions taken on that particular day, based on the outdated previous experience, are wrong. Nevertheless, as in the gradually changing world experiments discussed in the previous section, the memory limited configurations gradually converge to the performance of the knowledgeable policy. Again, smaller memory sizes lead to faster convergence while the configuration with infinite memory ($H = inf$) is problematic.

It is important to observe that all H=12 configuration with reset logic converge faster than H=8. This is because, after the mission on day 21, they promptly detect the anomalous performance (very large penalties due to wrong decisions) and reset the experience memory,

so on day 22 they start learning from scratch. The difference vs $H = 8$ is significant in the $0.2/0.8 \rightarrow 0.8/0.2$ scenario where previous experience is even more deceiving compared to $0.1/0.5 \rightarrow 0.5/0.1$. In the first case, all reset methods perform equally well, while in the second case LOF leads to slightly faster adaptation than IST, IQR and LOF+IST.

4.7.7 Estimated probabilities

For the heuristic, in all subsequent experiments, we use the configuration with H=12 and the LOF+IST option. Note that, based on the results presented above, there is no clear winner among the different reset methods. On the one hand, in a gradually changing world one would prefer a more aggressive anomaly detection variant. On the other hand, if such changes are rare or large/abrupt, it would be better to use a variant with a more conservative anomaly detection. To have a balanced solution that achieves good performance in both stable and changing worlds, we use a rather small experience memory in combination with the stricter anomaly detection method.

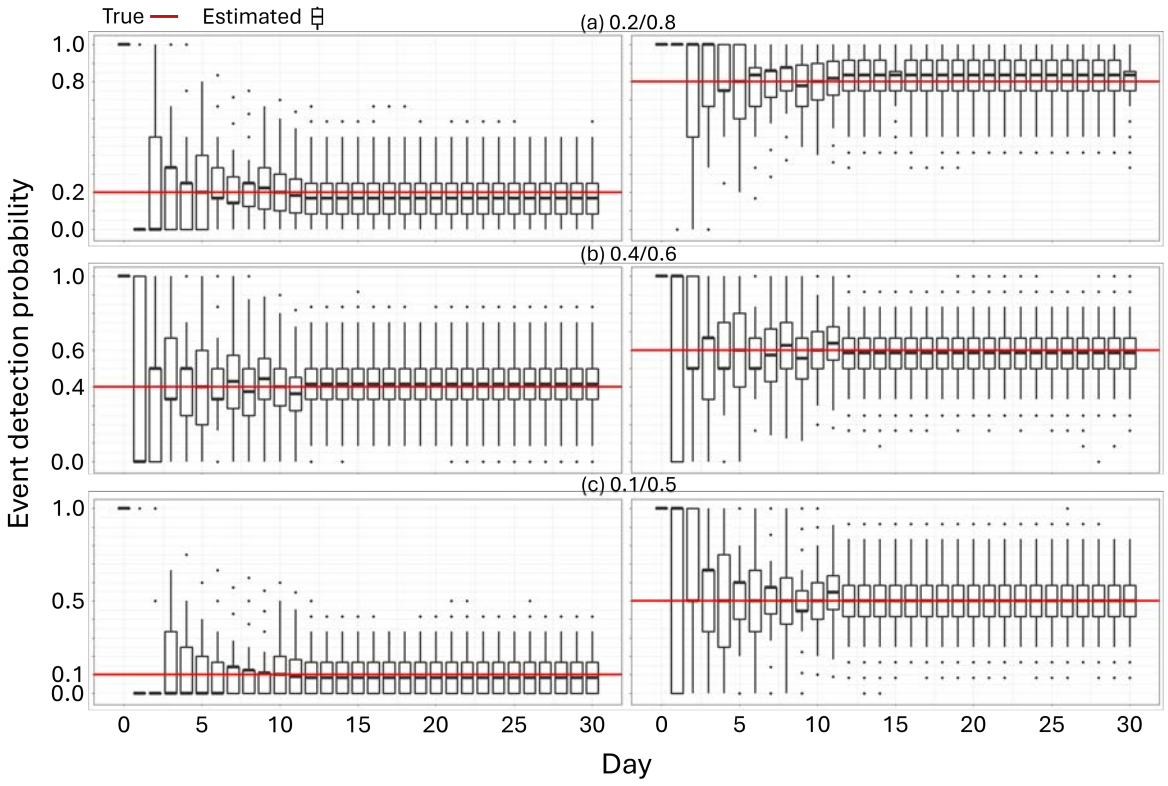


Figure 4.8: Probabilities estimated in the stable world by the chosen heuristic configuration ($H = 12$ and LOF+IST option). Row (a) corresponds to scenario 0.2/0.8, row (b) to scenario 0.4/0.6 and row (c) to scenario 0.1/0.5.

Figure 4.8 presents the probabilities estimated by the chosen configuration of the heuristic

in the stable world. Each row corresponds to a different probability scenario, while the left and right columns correspond to the points of the green and red regions respectively. The red lines indicate the true probability that holds for each scenario/region pair. Each boxplot corresponds to the probabilities estimated for the waypoints of the respective region, on each day for all 20 random scenario traces. The horizontal line inside the boxes is the median value.

Note that on day 0, the probability estimated in all waypoints is 1.0, as in the first day the drone always chooses to wait (default policy). As expected from the previous experiments, the median estimated probability (but also the size of the boxes) converge around day 12 in all scenarios. However, although the median of the estimated probabilities is generally close to the true probability, we can observe that there is some deviation in the estimations of the probabilities in some points. This can be observed by looking at the values of 1^{st} and 3^{rd} quartiles of each box, but also at the points (estimations) that fall outside the box (the vertical lines). Even with these deviations, the proposed approach performs close to the knowledgeable, given the results of the previous experiments. This is because as long as the estimated probabilities lead to the same decisions about whether to wait or go, as per Equation 4.3, the performance does not degrade. The relative increase of the mission time achieved by the heuristic vs the knowledgeable for the $0.2/0.8$, $0.4/0.6$ and $0.1/0.5$ scenarios is 0.7%, 0.7% and 0.9% respectively.

This validates that regardless the deviation shown in Figure 4.8, in most cases the heuristic takes the same decisions as the knowledgeable, while in some (outlier or almost outlier) points the heuristic will actually take the wrong decision (as per the true probability). It is more likely to take wrong decisions on scenarios where the true event detection probability is close to the point that either decision seems equally right, according to Equation 4.4. This is the case for the regions where the probability is 0.4, Figure 4.8(b), and 0.5, Figure 4.8(c). However in the former case, the right decisions are marginally beneficial than taking the wrong decision, hence there is no significant degradation in the performance of the heuristic in the $0.4/0.6$ scenario. This is explained in the next subsection in more detail. In the $0.1/0.5$ scenario, the inaccurate probability estimations affect the performance a little bit more, leading to a relative increase vs the knowledgeable of 0.9% (vs 0.7%, that holds for the other scenarios). Still, the performance drops negligibly for the same reason. Although the estimated probabilities may cross the point that lead the drone to take the wrong decision, since the true probability is

close to that point, the outcome of taking the wrong decision becomes less significant.

4.7.8 Heuristic vs the wait, go and random policies

In this part of the evaluation, we compare the heuristic vs the wait, go and random policies introduced in Section 4.7.2. Again, we report the relative increase of the respective mission times using as a reference the time achieved by the knowledgeable policy.

Stable world

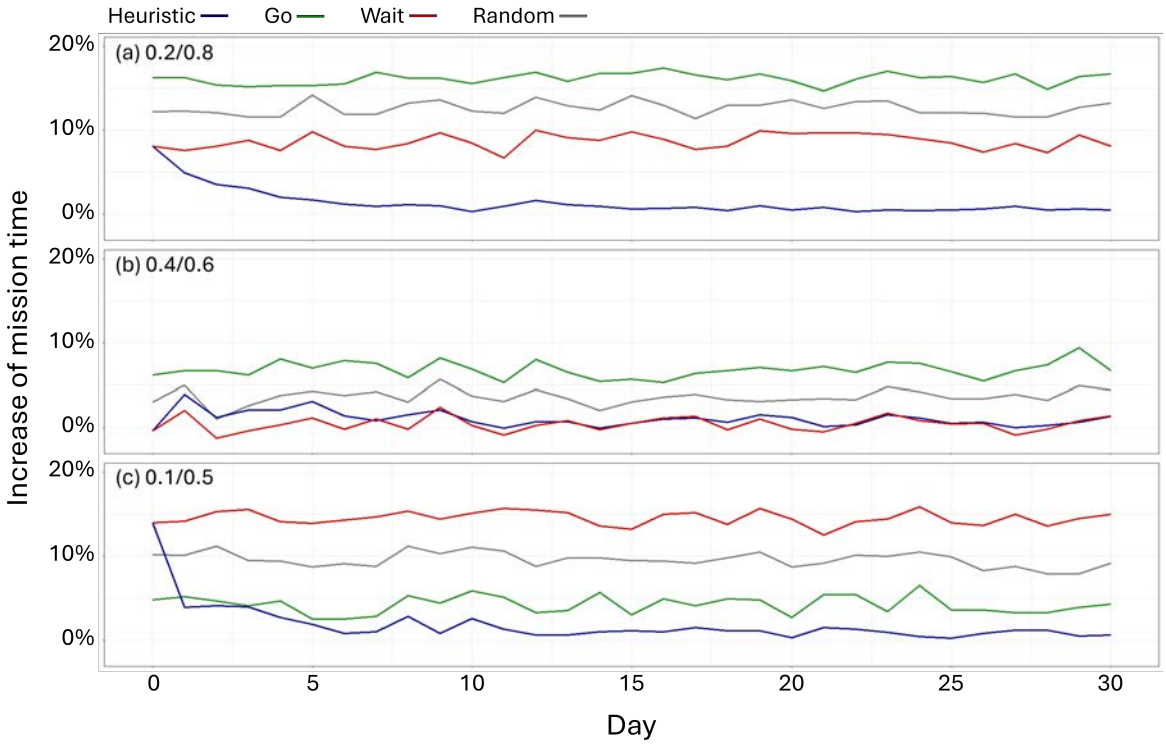


Figure 4.9: Heuristic vs the wait, go and random policies in the stable world.

The first set of experiments focus on the stable world, for each of the regional event detection probability scenarios in Table 4.1. The results are given in Figure 4.9. We see that in the $0.2/0.8$ and $0.4/0.6$ scenarios, the wait policy consistently outperforms the go policy. This is because, even though in the one region the best decision is to go and in the other the best decision is to wait, the penalty for wrongly waiting for the computation to finish is smaller than the penalty for wrongly going and having to return to the previous point of interest. Therefore, in both scenarios, the overall best (static) policy is to wait. This is not the case in the $0.1/0.5$ scenario, which is highly asymmetrical in terms of the event detection

probability of the two regions. Namely, in the 0.1 region the best decision is clearly to go, while in the 0.5 region the best decision is to wait but only marginally, given that the decision to go is as likely to be correct as to wait but the penalty for wrongfully going is larger than the penalty for wrongfully staying. Looking at the random policy, we can see that it always performs between the wait and go. This is expected since half of the wrong decisions are to wait and half of them are to go.

The heuristic clearly outperforms all three policies in the $0.2/0.8$ and $0.1/0.5$ scenarios. It performs close but slightly worse than the wait policy in the $0.4/0.6$ scenario, where wait leads to an average increased mission time vs the knowledgeable policy of only 0.5% while the heuristic achieves 0.7%. Note that, in this case, the wait policy performs close to the knowledgeable policy, which means that go decisions are generally wrong or marginally beneficial at best. While this is quite obvious for the 0.6 region, it is less obvious for the 0.4 region. In this case, even though the knowledgeable policy chooses go, this decision leads to very small gains in mission time compared to waiting at each point of interest. The reason is that the penalty of a wrong go decision (time needed for the drone to return to the previous point of interest) is larger than the penalty of a wrong wait decision (waiting at the point of interest for the computation to finish), almost canceling the fact that it is (slightly) less likely to have events detected in the 0.4 region. Overall, the accumulated penalties due to wrong wait decisions are negligible, hence the very small difference of wait vs the knowledgeable policy.

Gradually changing world

Next, we focus in the gradually changing world. As in the respective experiments for the different configurations of the heuristic, we investigate the $0.2/0.8 \rightarrow 0.8/0.2$ and the $0.1/0.5 \rightarrow 0.5/0.1$ scenarios. The results are shown in Figure 4.10. Looking at the first scenario, we observe a similar pattern for the wait, go and random policies. Namely, during the change period, performance initially gets better approaching that of the knowledgeable policy, after a while it gets worse, and when the change period ends it becomes the same as before the change. The initial but temporary improvement is because in the first part of the change period the smaller regional event detection probabilities gradually increase and the larger probabilities gradually decrease toward the point where it is practically indifferent whether to stay or go (both decisions are equally good). Once this point is reached, the

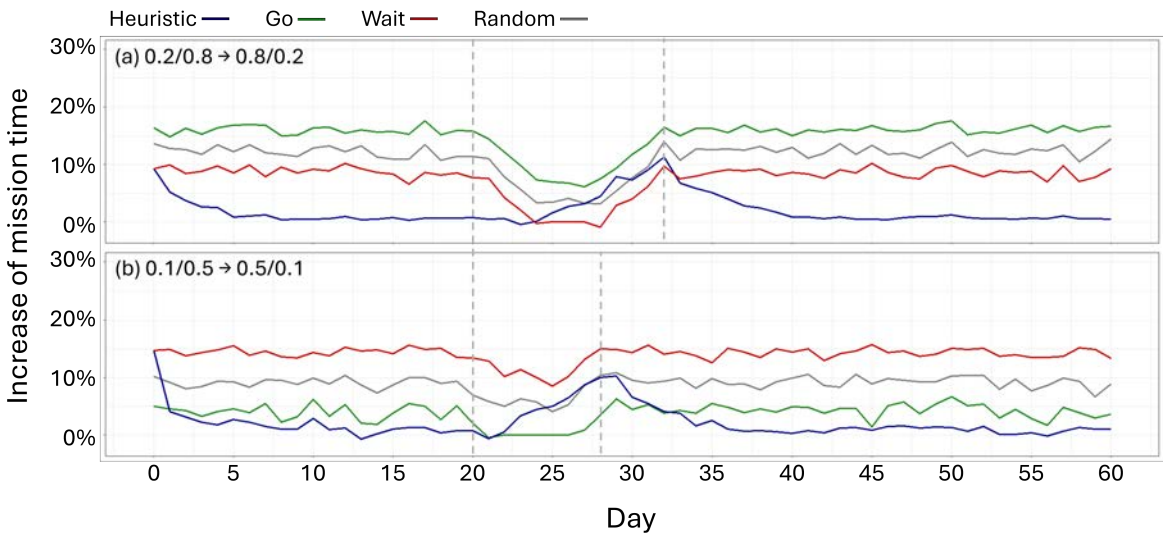


Figure 4.10: Heuristic vs the wait, go and random policies in the gradually changing world.

opposite trend begins as regional probabilities become smaller/larger thereby increasingly favoring a stay/go decision, respectively. When the change period ends, the regional probabilities are completely reversed and all policies naturally exhibit the same performance as before the change started. This pattern is also observed in the second scenario, albeit with a smaller performance improvement during the first phase of the change period due to the asymmetry of the regional event detection probabilities. In both scenarios, during the period of change, the heuristic performs worse than the best static policy in each case (wait in $0.2/0.8 \rightarrow 0.8/0.2$ and go in $0.1/0.5 \rightarrow 0.5/0.1$), but after the change it quickly converges to its previous performance outperforming all other policies.

Abruptly changing world

Finally, we investigate the same scenarios in the abruptly changing world. The results are shown in Figure 4.11.

We observe that the performance of the wait, go and random policies remains practically the same during the whole duration of the experiment. This is because the change is done abruptly, without any transition period. As in the previous experiments, the fact that the two regions end-up with reversed event detection probabilities does not affect the performance of these policies since the number of points of interest for which the right decision is to go or to wait remains exactly the same. The heuristic performs worse than any of the static policies on the day that immediately follows the change. As discussed in Section 4.7.6, this is because

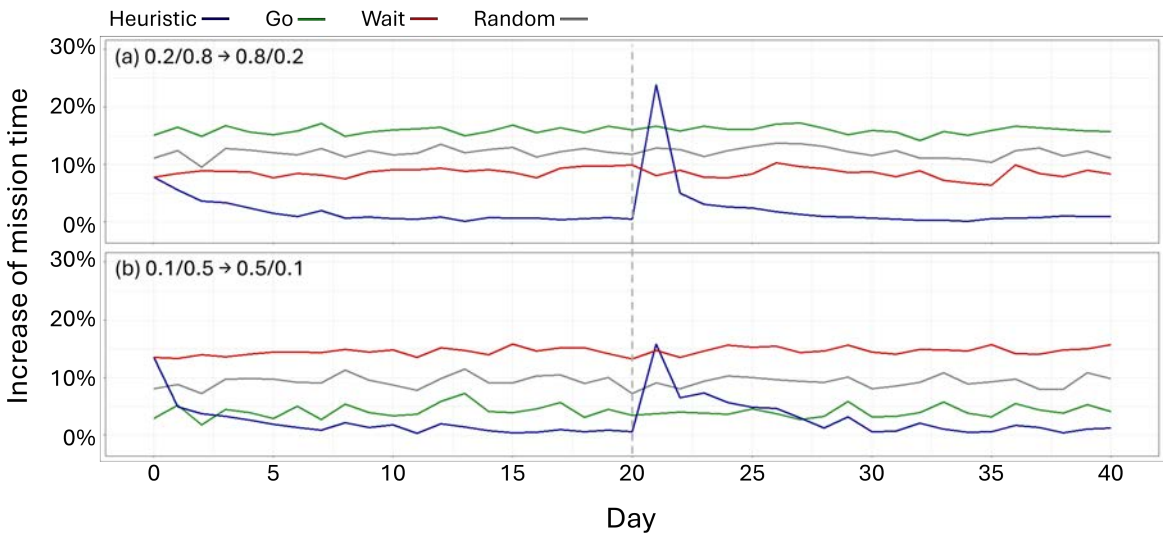


Figure 4.11: Heuristic vs the wait, go and random policies in the abruptly changing world.

it has completely wrong knowledge. However, after this day of very bad performance with severe penalties due to wrong decisions, it resets the experience memory and quickly adapts to the new reality outperforming all policies latest after the 7^{th} day in the changed world.

4.7.9 Heuristic vs the stay, go and random policies for different computation times

As discussed in Section 4.5.3, the time $procT$ it takes to process the sensor data in order to detect events, determines the penalty of a wrong wait or go decision, $procT$ and $returnT(wp_i, wp_{i+1}, procT)$, respectively. Thus, it affects the decisions that are taken by the heuristic (but also by the knowledgeable policy) according to Equation 4.3 and Equation 4.4. To explore this aspect, we perform a series of experiments where we compare the heuristic with the wait, go and random policies for the stable world and the regional event detection probability scenarios in Table 4.1, for different values of $procT = 8, 10, 12$ seconds.

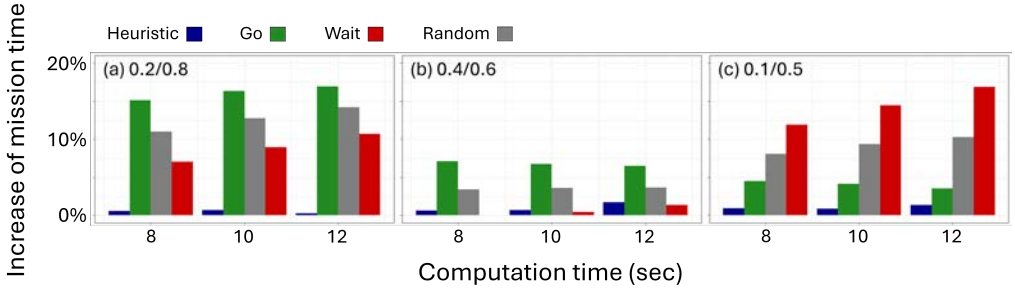


Figure 4.12: Heuristic vs the wait, go and random policies for different computation times.

Figure 4.12 shows the average performance from day 12 to day 30 where the heuristic has full experience memory (recall that we use the configuration with $H = 12$ and LOF+IST). The heuristic performs by up to $35x$, $56x$ and $47x$ better than the wait, go and random policies, respectively. While in absolute terms, the heuristic can save up to 3.5, 4.3 and 3.6 minutes of mission time vs the wait, go and random respectively.

The $0.4/0.6$ scenario is the only case where the heuristic is consistently outperformed even if slightly by a static policy, namely wait, for the reasons already discussed in the previous experiments. In fact, for $procT = 8$, the knowledgeable policy behaves like the wait policy, deciding to wait in all points of interest, as can be inferred from the 0% increase in the mission time of wait. As discussed previously, this is because the penalty of a wrong go decision is larger than the penalty of a wrong wait decision, outweighing the fact that event detection is slightly less likely in the 0.4 region. Note, however, that this balance gradually changes in favor of go decisions for larger values of $procT$, as can be seen from the slightly growing mission time of wait for $procT = 10, 12$. The reason for this somewhat unexpected behavior, is that the penalty of a wrong go decision grows slower than the penalty of a wrong wait decision, namely $\frac{returnT(wp_i, wp_{i+1}, procT)}{procT}$ is $1.56, 1.44, 1.33$ for $procT = 8, 10, 12$, making go decisions more attractive as the processing time increases.

A general observation is that the performance tends to drop when increasing the computation time. This is expected since larger computation times translate to larger penalties for wrong decisions. As an exception, in the $0.4/0.6$ scenario and even more so in $0.1/0.5$ scenario, the go policy performs slightly better for larger values of $procT$. As noted above, the go decision becomes more attractive as the penalty for wrongly going gets closer the penalty for wrongly waiting as the processing time increases. Also, in the 0.6 and 0.5 regions with the highest event detection probability in these scenarios, the go decision is much more likely to be correct than in the 0.8 region in the $0.2/0.8$ scenario. Another exception can be found in the $0.2/0.8$ scenario for $procT = 12$, where the heuristic performs better (closer to the knowledgeable policy) than for $procT = 8, 10$. As explained above, the penalty of a wrong go decision grows slower than the penalty of a wrong wait decision for larger $procT$. Thus, the decision of the heuristic may tip in favor of go at points of interest where wait was previously (for smaller values of $procT$) a marginally better choice. More specifically, this is the case for some points of interest in the 0.2 region where the (imperfect) probability estimate of the heuristic is considerably greater than the true underlying event detection probability.

Chapter 5

Autonomous Drone-Based Sensing System

5.1 Introduction

As drones are increasingly adopted across a growing range of applications, the need for reliable and efficient drone operation becomes critically important. Existing APIs and tools [2], [85] can contribute to the design and implementation of programs that automate drone-based missions, minimizing or even completely eliminating the need for direct control by a human pilot. This is particularly important if the mission has to be repeated frequently, which is typically the case in several monitoring and surveillance applications. It also allows a tighter monitoring and analyzing of the drone's status as well as a truly data-driven mission execution. At the same time, different technologies can pave the way towards the full automation of the entire drone operation cycle. For example, available precision landing sensor technologies allow a drone to accurately land in small target areas, such as a landing pad of a hangar, which can be used to shelter the drone when it is not being used. Charging technologies can be used to charge the drone's batteries without having to manually plug the drone's battery to an electrical interface. Also, on-site weather stations can provide information on the ability of the drone to currently execute a mission or not, while a nearby server or ground station can help with the (online or offline) computation of the data gathered by the drone(s) to produce information, which can be used to guide the rest of the mission or can be sent upstream to other systems and/or the end-user.

In this chapter, we present a complete architecture for a system that integrates multiple

drones, their hangars and battery charging subsystems, a local weather station and all the software components needed to provide a complete and fully autonomous remote sensing service. The main contributions are: (i) We propose a modular architecture for autonomous drone-based remote sensing systems, which can operate at the edge in a standalone way or as part of a more complex ecosystem. (ii) We describe the proposed design in detail, focusing on the software components that are responsible for the core system operation and mission execution, providing support for the full drone operation cycle. (iii) We discuss the drone and ground station setup we use in our field tests as well as a simulation environment used to test the functionality of our system in the lab for a wide range of scenarios.

We wish to note that our work focuses on outdoor sensing scenarios, using drones with autonomous flight, navigation and landing capability – many such platforms exist, and nowadays it is also relatively straightforward to even have custom designs, such as the drone we use in our tests. While some elements of the proposed system could be potentially reused to support indoor scenarios, this direction is beyond the scope of this work.

The rest of the chapter is structured as follows. Section 5.2 gives an overview of related work. Section 5.3 presents the high-level system architecture. Section 5.4 discusses the main control logic, while Section 5.5 focuses on the aspect of mission execution. Finally, Section 5.6 describes the hardware-based and simulation-based setup used to test our implementation.

5.2 Related Work

There is a growing body of work on supporting the development of autonomous drone-based monitoring and tracking applications. Below, we briefly discuss indicative efforts. A modular system for object-tracking applications is presented in [108], consisting of a drone with an onboard computer, a camera and positioning sensors like GPS. The camera images and sensor data are processed on the onboard computer in order to detect the target object and infer the drone's position. The results fed in a mission planning module, also running on the onboard unit, which sends suitable control commands to the drone's flight controller so as to follow and approach the target object. [109] describes a drone-based system for structural health monitoring. The drone is equipped with a camera and an onboard computer that processes the camera images to detect special AR tags, which are used to mark the desired

points (structural elements) of interest. It also features a separate stereo vision subsystem with two special (CCD point-grey) cameras. The drone flies autonomously to the points of interest where it takes stereoscopic images. After the flight, these images are transferred to a high-performance computer in order to perform the required data processing. The goal of the system presented in [110] is to pursuit intruder drones based on external alerts. When an intruder drone enters the area, another drone is used to detect, track and jam the intruder. The drone is equipped with a camera, an onboard unit that processes the camera images in order to detect the intruder drone and guide the drone by sending signals to the flight controller, and a software-defined radio used to jam the intruder as well as for self localization. The authors in [111] propose a system for precision agriculture using UAVs and an edge server. The photos taken by the UAVs are processed on an edge server to extract relevant features for the crop in question, and to decide if more samples are needed, in which case the next flight plan for the UAVs is produced. The above efforts focus on the autonomous operation and flight control of the drone, in some cases with the help of edge computing infrastructure. Our work is complementary to such efforts, with the goal to automate the full operation cycle of drone-based sensing systems, including storage, battery-charging, weather checking, mission planning, mission execution and data processing, without any human intervention. The proposed design can be flexibly customized to support different specific application scenarios, including the ones outline above.

There are several efforts on supporting the coordinated usage of multiple robots or drones. TeCoLa [112] abstracts autonomous robots as nodes providing different services that can be accessed remotely via method calls with at-most-once semantics. Mobility is modelled as a special service. The application program can transparently discover and invoke the available nodes as needed, while there is also support for the dynamic formation and grouped invocation of smaller teams. Resh [113] is a domain-specific language and orchestration system for multi-robot systems. Each robot has an agent that communicates with a centralized runtime in order to advertise its capabilities/status and translate the Resh protocol to the local robot-specific API. Dolphin [114] also follows a centralized approach for task execution using a static team of autonomous vehicles, where the platform's runtime system is responsible for polling the vehicles, sending tasks to them and managing platform-specific operations. In [115], the ScaFi programming language is used to let a multi-agent system achieve a collective goal in a distributed fashion. An iterative protocol is followed, whereby each agent

evaluates its local state, checks for any messages coming from its neighbors, runs an aggregation program based on the current context, performs corresponding actions, updates its state and notifies its neighbors. Maple-Swarm [116] allows the developer to submit partial high-level plans. These plans are used as input to generate specific tasks for one or more robots, based on their capabilities and the current state of the system. Our work is largely orthogonal to the problem of coordinated task execution. Our current implementation uses TeCoLa for mission execution, which provides support for team-level operations that can be used to support sensing tasks with multiple drones. However, in principle, any other platform could be used to write the driver program for executing such sensing tasks.

Some researchers investigate the remote usage of UAVs following cloud-based approaches. In [117], UAVs appear as cloud services managed by a central coordinator, which takes care of the underlying communication and maintains the sessions between UAVs and their users. [118] proposes a cloud service through which users can interactively request missions using an Android application. The service plans the path, picks the best available UAV and sends the mission to the UAV's autopilot, while the user can monitor the mission by receiving live status data and video streaming. Dronemap Planner [119] virtualizes UAVs by offering suitable control and data retrieval web services, which run in the cloud and can be invoked by third-party applications via standard web-service protocols. All these approaches are definitely interesting if ones wishes for UAVs to become a shared resource that can be used by different users possibly even at the same time. Our work goes in the opposite direction, focusing on the automation of remote sensing systems that employ their own (possibly highly customized) drones and ground infrastructure in an exclusive way. Also, both control and data processing take place at the edge, making it possible for the system to operate with full autonomy even when having poor connectivity with the cloud. If needed, the proposed system can also work in the cloud, using a 5G link for the communication with the drone, provided the latency and bandwidth are sufficient for the application in question.

Our work has some common goals with the concept of autonomic computing [120]. The benefits of an abstract architectural approach for autonomic computing systems are stressed in [121], where the authors motivate their model inspired by the domain of robotics. Most work in this area of research focuses on resource/service monitoring and provisioning in cloud environments so as to maintain QoS and satisfy SLA requirements [122] [123] [124], while several self-* properties have also been studied in the context of flying ad-hoc networks[125] [126].

In contrast, the approach proposed in this paper tackles the self-management of drone-based remote sensing systems, by monitoring key parameters of system operation and taking corrective or preventive actions as needed.

5.3 System Concept and Architecture

We assume a geographical area or specific points of interest where certain sensing tasks can be performed using one or more drones. On the one hand, there can be standard sensing tasks that have to be conducted on a periodic basis, without any explicit request. On the other hand, certain sensing tasks can be introduced on demand, triggered by requests from external systems, such as alarms or end-user applications.

To support such a remote sensing capability, we propose a system that can manage the full cycle of operation — not just the flight of the drone during a mission, but also all the take-off and landing, storage and battery charging procedures — with little or no human intervention. Of course, the sensing equipment that needs to be mounted on the drone, the data collected via these sensors and the processing that needs to be performed on this data, all depend on the application. Nevertheless, having a general system design and implementation, which properly addresses all of the application-neutral aspects of such drone-based remote sensing system, is of big value as it is possible to support any specific application with minimal customization and debugging effort.

In the spirit of edge computing, we envision such a system to run on a ground station close to the area of operation. This way one can have a direct and low-latency communication with the drone via WiFi or long-range RF technology, enabling autonomous and robust operation without relying on good connectivity with the cloud — note that, even in the era of 5G, fast and stable Internet connectivity is by no means guaranteed if the system should be deployed in remote areas. The basic system infrastructure includes suitably designed hangars where the drones can land autonomously, battery charging subsystems (inside each hangar) and a local weather station, with which the ground station can communicate over dedicated serial links or a local area network. We assume that standard GPS accuracy is sufficient for the navigation and sensing tasks of the drones. Finally, we assume that the drones can land on their hangars with high accuracy, supported by an appropriate precision landing mechanism, and dock on the battery charging system without manual intervention.

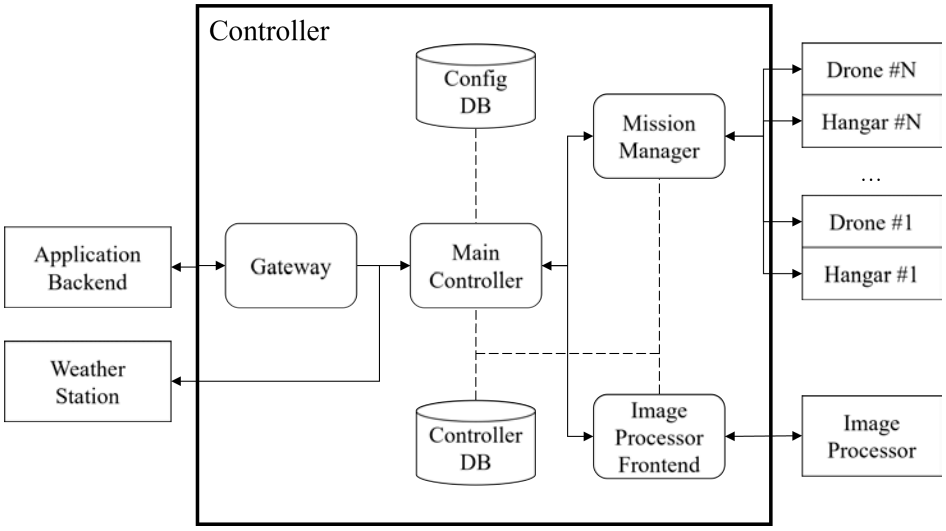


Figure 5.1: Top-level system architecture.

From a software perspective, the proposed system is built in a modular way, being composed of micro-services which are combined to provide the overall functionality. The top-level system architecture is shown in Figure 5.1. Some components play the role of high-level drivers and provide access to the physical parts of the system, while others are responsible for purely software-based functions. Next, we outline the role of each component.

The Gateway provides a technology-neutral interface towards external systems, like the end-user application through which the system can be managed remotely. Its role is to convey incoming requests to the Main Controller and send back the corresponding replies. It is important to stress that the interaction with external systems is not crucial for the core system operation; the system can execute monitoring tasks in an autonomous manner, even when having unstable/bad connectivity to external systems. Notably, access to the Gateway needs to be properly secured in order to avoid unauthorized access. This can be achieved using well-established technologies, such as VPNs in combination with more refined access control mechanisms (which is not the focus of this work).

The heart of the system is the Main Controller, which is responsible for the automation of the entire cycle of operation. Briefly, the Main Controller selects or creates a flight plan, performs the necessary checks (such as checking weather conditions), submits the plan for execution and monitors its progress until completion. At the end of the execution of a flight plan, the Main Controller processes the flight data. Section 5.4 describes the Main Controller in more detail.

The weather station can be accessed through a suitable REST API. This is an external

subsystem that provides the Main Controller insight about the weather conditions in the target area.

The Mission Manager, manages the drones and executes the mission submitted by the Main Controller. This is done using a so-called driver program, which performs the required interactions with the drones. We consider that each drone has its own, separate hangar, also managed by the Mission Manager. This is described in more detail in Section 5.5.

The Image Processor Frontend can be invoked by the Main Controller through a suitable API. This front-end bridges the controller with the image processing subsystem which is invoked after a mission is complete.

Further, the system keeps certain information on persistent storage. This includes specific points of interest, no-fly zones and drone characteristics, e.g., estimated flight time as a function of average flight speed, which may need to be taken into account when generating a new mission plan. Also, the system stores the available mission plans along with the respective driver program for their execution. Last but not least, it keeps a log for each sensing task and corresponding mission execution, including its status and the data that was collected (and possibly processed) in this context. The different system components create/access this information via a shared repository, e.g., using a database for structured information in conjunction with a file system for unstructured data. This information can also be inspected and updated by authorized external systems and end-user applications via the Gateway.

5.4 Main Controller

The Main Controller oversees the different phases of a full monitoring cycle, in a similar way this would be done by a human operator. Namely, it invokes other components to perform the necessary actions and decides when to move the system to the next state of operation.

The internal organization of the Main Controller is shown in Figure 5.2. It includes an RPC Server that is invoked by the Gateway when external requests arrive, an event-driven finite state machine (FSM) that implements the coordination of the operation cycle, and a set of observers that monitor different processes or subsystems and issue an event when certain conditions apply. These entities are independent threads running concurrently to each other. Sensing tasks are kept in a queue, sorted according to the scheduled start times. The events generated by the observers are kept in a separate FIFO queue, from where they are retrieved

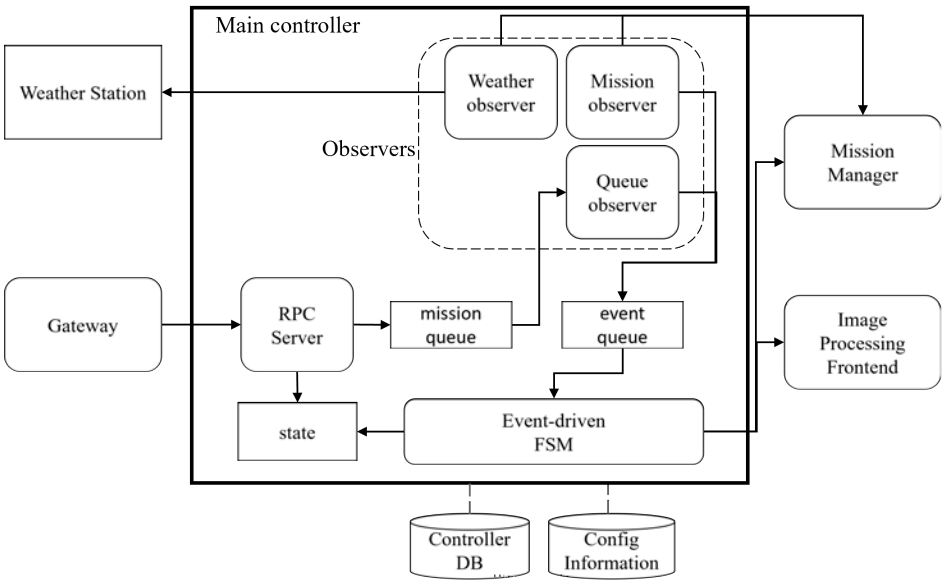


Figure 5.2: Internal organization of the Main Controller.

and handled by the FSM in a sequential manner. The operation of the FSM and the observers is described in more detail in the following subsections.

5.4.1 Finite state machine

The FSM runs in a loop. In each iteration, the next event is removed from the queue and the respective handler is invoked, which performs a state transition along with some actions. An event is dropped if it is not compatible with the current state or the last processed event was of the same type. If the queue is empty, the FSM blocks until an event is added to the queue.

The FSM has three states, representing the key phases of the operation cycle. The state diagram and the respective transitions are given in Figure 5.3. The events that trigger the state transitions are indicated by the capitalized labels over the edges while the sources of each event are given in bullets. Figure 5.4 shows the core logic of the event handlers, where the arrows denote the logical flow between handler invocations resulting as a side-effect of event generation.

Initially the FSM is in the READY state and indicates that the system is ready to perform a new sensing mission. The start of a new mission is signaled by the START_MISSION event through which the system transitions to the IN PROGRESS state that initiates and monitors the execution of the mission. Through the MISSION_COMPLETED event, the successful completion of the mission is signaled. This means that all of the drones used for this mission

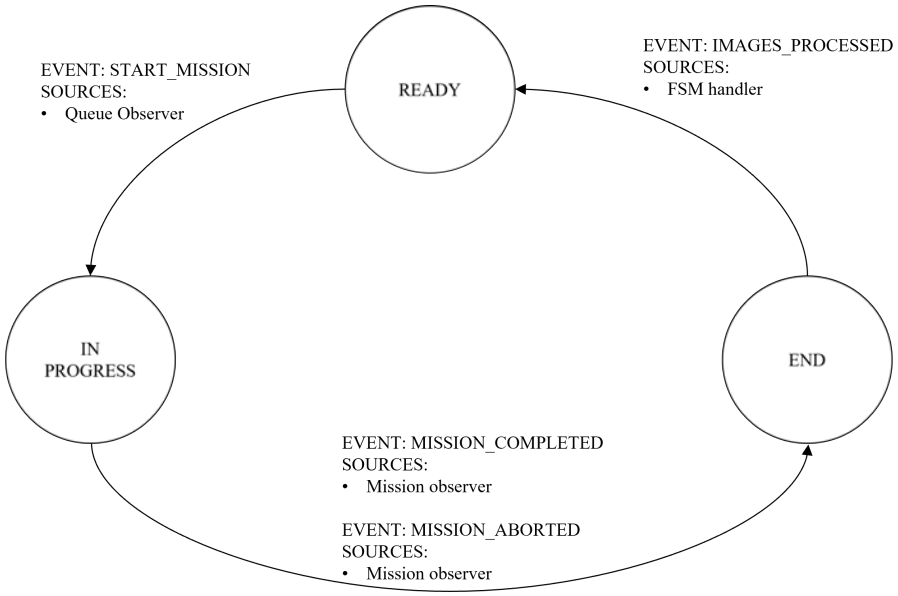


Figure 5.3: State diagram of the FSM.

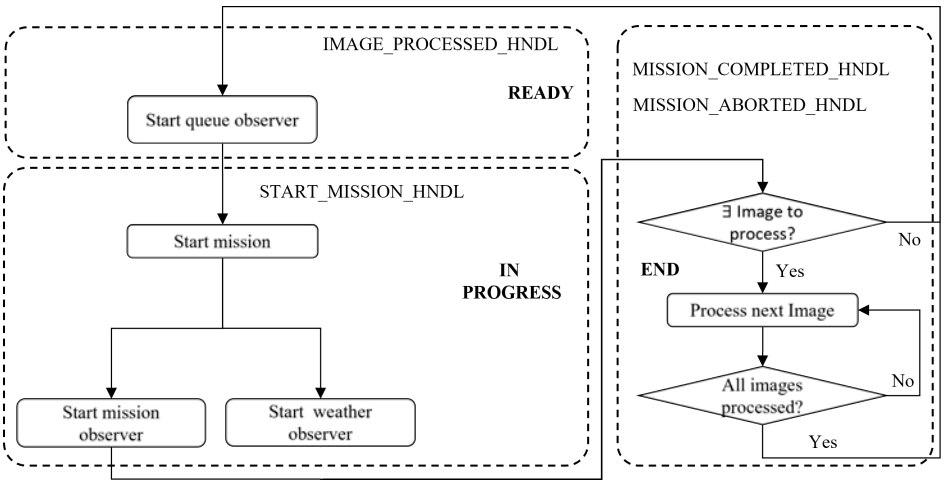


Figure 5.4: Event handling logic of the FSM.

have successfully landed on their hangars, and the system transitions from the IN PROGRESS state to the END state, where the data gathered are processed and stored.

The execution of the flight plan can be canceled for various reasons, while the system is in the IN PROGRESS state. The cancellation is signaled by the MISSION_ABORTED event, in which case the system transitions to the END state. This event is generated by the mission observer due to unsuccessful checks or failure to execute specific steps during the mission, due to the weather observer in case the weather deteriorates or due to the request of the user.

Finally, from the END state the system transitions to the READY state via the event IMAGES_PROCESSED, which indicates that the processing of the images has been completed

and the results have been successfully stored in the system database.

5.4.2 Observers

The observers are started on a need-to basis by the event handlers of the FSM, as indicated in Figure 5.4. An observer is automatically stopped when it generates an event or the FSM makes a state transition.

The Queue Observer is responsible for triggering the execution of the next sensing task. To this end, it checks the mission queue, determines when the next task needs to start and sets a timer accordingly in order to issue the START_MISSION event. In case a task is postponed or rescheduled (e.g. due to the weather, its execution was aborted), it is put back at the head of the queue so that its execution starts as early as possible.

The Weather Observer uses a suitable API to receive information about the weather conditions from the local station. If weather conditions deteriorate while a mission is already in progress, the Mission Manager is informed to abort the mission. Note that in such case, the FSM remains on the IN PROGRESS state. This will change only after the mission continues and is complete.

The Mission Observer periodically polls the Mission Manager in order to receive information about the mission execution status. It generates the MISSION_COMPLETED or MISSION_ABORTED events depending on whether the mission was fully and successfully completed or aborted.

5.5 Mission Manager

The Mission Manager component is responsible for actually running a sensing task according to its specific mission plan, while communicating with the drones as needed. The interaction with the Main Controller takes places through an RPC Server, while the core functionality of mission execution is implemented in a separate environment that is invoked by the RPC Server, via a suitable front-end that hides technology-specific details. In our current prototype, the mission execution environment is implemented using the TeCoLa platform [112], as shown in Figure 5.5. TeCoLa supports a flexible, service-oriented interaction with the drone in Python, and can work over different wireless technologies, including WiFi and long-range RF.

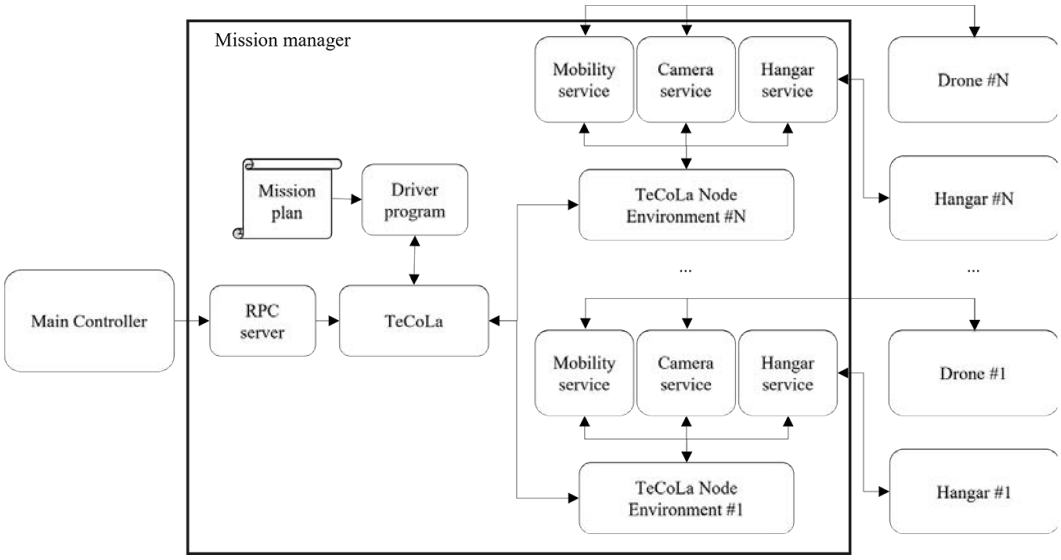


Figure 5.5: Mission execution approach.

In a nutshell, the Mission Manager takes as input a mission plan and the driver program that will parse and execute the plan by invoking the drone's service as needed. The driver program runs on top of the TeCoLa's mission execution environment, which takes care of the underlying communication with the drone over a wireless link. Each TeCoLa node environment, representing a different drone, processes the requests of the mission execution environment, calls the corresponding local services and sends back the replies. In the basic system configuration used to test our system, the node environment provides three services: the mobility service for the navigation-related functions of the drone's autopilot, the camera service for taking pictures via the drone's onboard camera and the hangar service for the interaction with the respective hangar.

5.5.1 Mission execution state

The state of the mission execution is kept in the TeCoLa mission execution environment, Figure 5.6. Initially, it is in the INACTIVE state where no mission is being executed and the system is ready to start the next one. When a command to execute a mission arrives, it starts the driver program with the flight plan as an input parameter and transitions to the RUNNING state. From the RUNNING state, it transitions to the COMPLETED state when the mission is successfully completed or to the ABORTED state if the mission is canceled. Finally, from the ABORTED or COMPLETED state, it transitions to the INACTIVE state after the finalization of the mission (all hangars are closed and the images captured are downloaded and

processed). In our work we also support commands to suspend and resume the mission. When the mission execution is in the RUNNING state it transitions to the SUSPENDED state when the command to suspend/pause the mission arrives, and back to the RUNNING state with the command to resume/continue the mission.

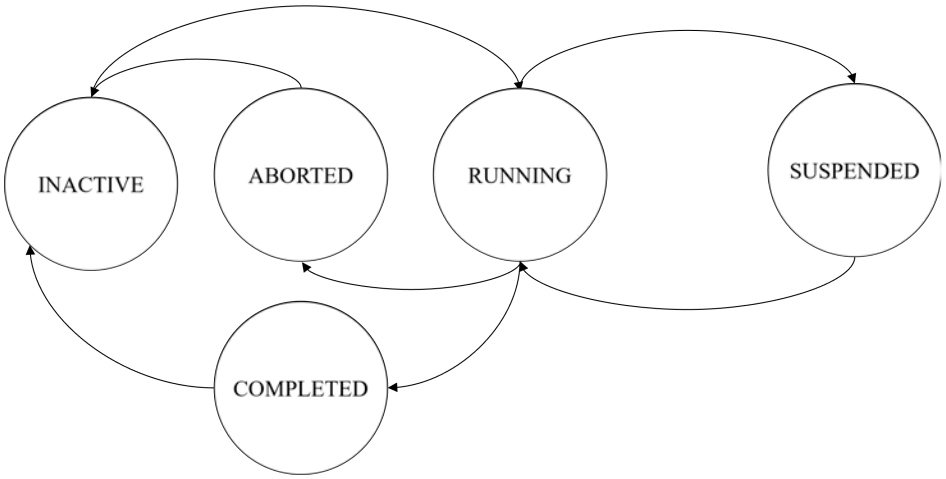
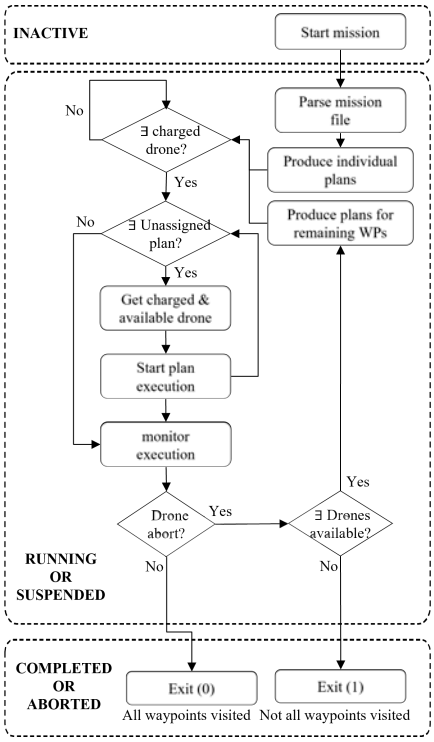


Figure 5.6: Mission state diagram.

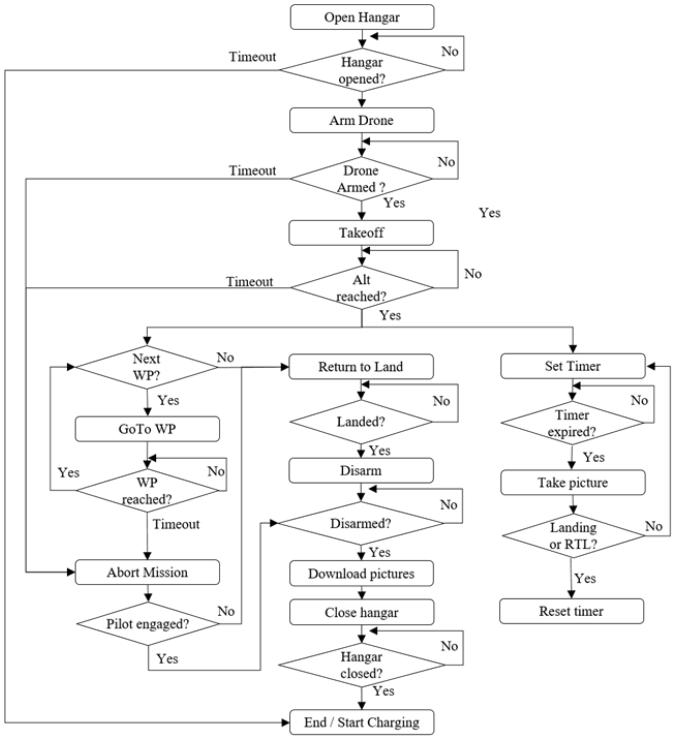
5.5.2 Mission execution

The driver program reads and interprets the flight plan file and accordingly controls the drones to execute it. The diagram in Figure 5.7a shows the operation of the driver program in combination with the states of the Mission Execution Engine. The driver program decides in how many individual plans the mission should be split into, based on the size of the mission and the autonomy of the system's drones, and generates corresponding flight plans. Then, when a drone is ready and charged and there is a plan that has not been assigned to another drone, the specific drone is activated and undertakes that plan. If all plans are completed successfully, the program terminates with a success code. Otherwise, if the execution of a drone's plan is canceled, the program removes the drone from the mission team and a new flight plan is created for the remaining waypoints assigned to the drone that made the cancellation, which is added to the list of pending plans to be executed by the next available drone. If there are no other drones available and there is still one or more plans pending, then the program terminates with a failure code as the overall mission was not completed with full success.

As an example of a driver program, Figure 5.7b illustrates the main logic of a program that is designed to execute waypoint-based mission plans while using the camera as a sensor.



(a) driver program execution flow.



(b) Main logic of a driver program for taking aerial images over a target area following a waypoint-based mission plan.

Figure 5.7: Mission execution core logic.

This focuses on one drone only. In the case that multiple drones participate in a mission, there will still be only one driver program responsible for all the drones. In other words, one driver program executing multiple such diagrams, one for each drone. Pictures are taken periodically during the entire mission as the drone moves between the waypoints, as well as at certain waypoints specified in the plan. Initially, the program arms the drone and instructs it to take-off. It also starts a timer in order to perform periodic sensing, shown in the figure as an independent thread. Concurrently, in a loop, the program instructs the drone to move to the next waypoint. If a sensing action is required at that point, the drone is instructed to take a picture before moving to the next waypoint. When the drone successfully visits all waypoints, it is instructed to return and land. Finally, the drone is disarmed, the recorded pictures are downloaded from the drone and saved in the system, and the respective hangar is ordered to close its doors.

If the drone does not reach the take-off altitude or a waypoint within a reasonable amount of time, this is considered as a failure either due to some external factors or due to some issue in drone's hardware or autopilot. To be safe, the driver program decides to abort the

mission, and instructs the drone to return to the hangar and land immediately. Similar checks can be performed for other things, such as the drone's battery level in order to abort the mission if this drops below a safety margin (not shown in Figure 5.7b to avoid clutter). If the drone fails to arm/disarm or does not land properly, is not possible for it to continue with its plan without manual inspection and perhaps even hands-on maintenance. In any of these scenarios, the drone will abort and be removed from the fleet. In such case, the remaining part of its assigned plan will be assigned to another (if any) drone (see Figure 5.7a).

At any point in time during execution, the Main Controller may ask the Mission Manager to pause, continue or abort the mission, due to a corresponding external request received via the Gateway. Also, the Weather Observer may request an abort if it detects deteriorating weather conditions. In these cases, the Mission Manager up-calls special handlers of the driver program, which take the necessary actions. More specifically, on pause, the last command that was sent to the drone is recorded and the drone is instructed to hover in its current position, while the driver program is suspended. When requested to continue execution, the last command is re-sent to the drone and the driver program is resumed. Finally, a requested abort leads to the same behavior as when the driver program takes this decision by itself. The event handling logic of the driver program is not shown in Figure 5.7b for brevity.

Note that the case that a sensing mission was aborted for some reason, is reflected in the return code of the program and the mission completion status reported by the Mission Manager to the Main Controller. Then the latter can identify and reschedule such aborted missions when the issue that led (e.g. weather conditions, user's command) to the abortion is resolved.

We wish to stress that the format of the mission plans as well as the driver program used for their execution can both be customized according to the application's requirements. It is also straightforward to introduce additional sensor (and actuator) services, provided the drone features the corresponding hardware. Furthermore, the system can employ different types of missions and corresponding driver programs, which may rely on platforms other than TeCoLa; one merely needs to a add a corresponding mission execution environment and front-end. This flexibility is important for sensing applications where there is no single "size" that fits all needs. In particular, it is possible to support dynamic data-driven missions. This can be done using a suitable driver program, which receives data from the drone during the mission (e.g., at specific waypoints), processes this data and guides the drone based on

the results. To minimize delays, heavyweight data transfers between the drone and the driver program on the ground station would need to be performed over sufficiently fast wireless links. The driver program can perform the required data processing by invoking in a direct way the appropriate services via the Processing API, without involving the Main Controller. Depending on the system configuration and available computing resources, the data processing services can run locally on the ground station or at a powerful edge infrastructure with good connectivity to the ground station.

5.5.3 Flight plans

Flight plans for missions where the drones must perform the sensing task on specific waypoints can be generated using algorithms like the one proposed in Chapter 2. However there are also missions where the drones must perform a sensing task across an edge/path. On the one hand we could adapt the previous mentioned algorithms by introducing constraints in the visitation order of the waypoints. For example introduce the hard constraint that the nodes that compose an edge, must be visited one after the other. On the other hand we could use an algorithm designed for the arc routing problem [127]. This is a routing problem that focuses on applications where the service is performed across arcs (edges), rather than on separate nodes/locations.

Note that the plan generation occurs inside the driver program which can be customized. One can easily replace the planning algorithm to match the needs of a specific mission according to its type and objective. Additionally, the size of a mission may require more sophisticated algorithms. For example, for small missions a simple and fast greedy heuristic, can perform well, while for larger missions a more complex algorithm, with wider exploration capabilities, may be required.

5.6 Functional Testing

We have implemented a complete system prototype, where all the software components discussed in the previous sections are separate microservices, packaged as Docker containers in order to support a managed deployment on the ground station. The mission plans used to test our implementation are waypoint-based, and the logic of the driver program that executes such plans is similar to that in Figure 5.7b, with several additional checks that may lead

to an abort or failure depending on the gravity of the detected issue. We have performed a wide range of tests using custom drones and ground-station but also a suitable simulation environment. We describe the two setups in more detail below.

5.6.1 Simulated setup

While field trials are important to verify the desired system operation under real conditions, they also come with several limitations, primarily due to safety rules and regulations in drone operation. Field trials are also very time consuming since they require extensive preparation and travel. For these reasons, a large number of more complex experiments are performed using a simulated setup described in the sequel.

To thoroughly test all system functions and several corner cases in a flexible and controlled way, we use a modular setup, which integrates the SITL emulator [85] of the autopilot, which is popular in the community along with container technology in order to support experimentation with different virtual unmanned vehicles and virtual ground stations. The setup is illustrated in Figure 5.8 and is described in more detail below.

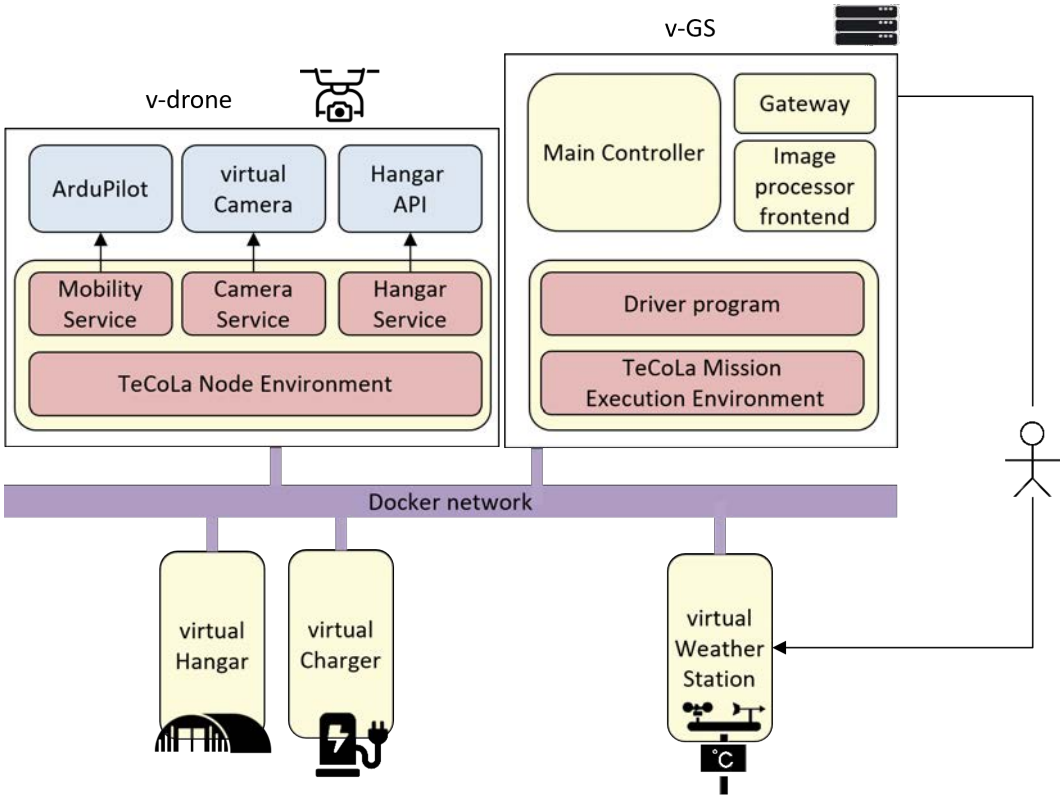


Figure 5.8: Simulated system setup.

The drone functions are implemented as shown in the v-drone schema, mimicking the

desired drone configuration. The TeCoLa Node Environment runs within a Docker container [84] while the mobility, camera and hangar services are implemented as wrapper methods implementing the corresponding API calls. In this setup, we employ the official software-in-the-loop (SITL) configuration of Ardupilot [85]. This runs as a separate process connected with the mobility service via the Dronekit library [2], in the same way this is done in the real drone. The camera service is modified to access pictures (based on the drone's current position in the virtual world) that are stored in a directory captured from previous flights. The hangar service communicates with a mockup hangar (explained bellow).

The ground station (v-GS) also consists of Docker containers, composing the entire management system. More precisely, each of the Main Controller, Gateway, Image Processor and the TeCoLa Execution Environment run each inside a different container. For the weather station, the hangar and the battery charger, we use mockups which also run as independent containers and are accessed by the corresponding API components. These mockups can be pre-configured as well as interactively controlled by the user during the simulation to behave according to the test scenario. More specifically, the weather station can report different conditions, while the hangar can open/close its hatch and the charger can appear to charge the drone's battery with a configurable delay or exhibit a malfunction. The communication between each component and simulated subsystem in the system setup happens over a local network created by the Docker environment bridging the different containers.

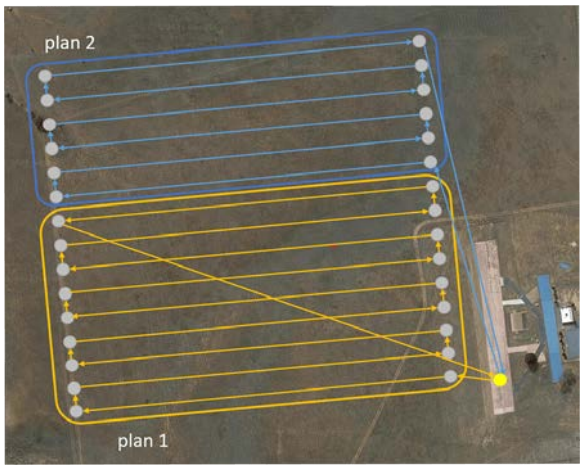
5.6.2 Indicative test scenario - single drone

Using the simulated setup, we have tested our implementation extensively for a wide range of scenarios regarding different cases of user intervention (mission pause, continue and abort), dynamically changing weather conditions, failures of the hangar/chargers and disconnections between the ground station and the drone. To give an indicative example, assume the administrator has added a periodic sensing task for scanning a target area of 300×300 meters through several passes at an altitude of 10 meters. The desired path of the task is defined via a sequence of waypoints with a horizontal distance of 300 m and a vertical distance of 20 m, depicted in Figure 5.9a by the gray bullets.

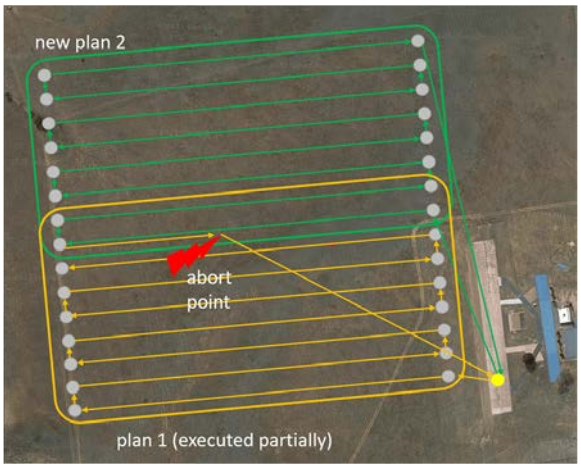
The total flight distance that needs to be covered is about 5 Km without taking into account take-off and landing, at a specified flying speed of 4 m/s. However, at that speed, the drone has an autonomy of only about 3.5 Km. As a consequence, this task is mapped to two smaller

mission plans, illustrated in Figure 5.9a in orange and blue color, respectively. Since this is a periodic task, these plans are produced offline and are stored in the system repository from where they are retrieved by the Main Controller when the task is started.

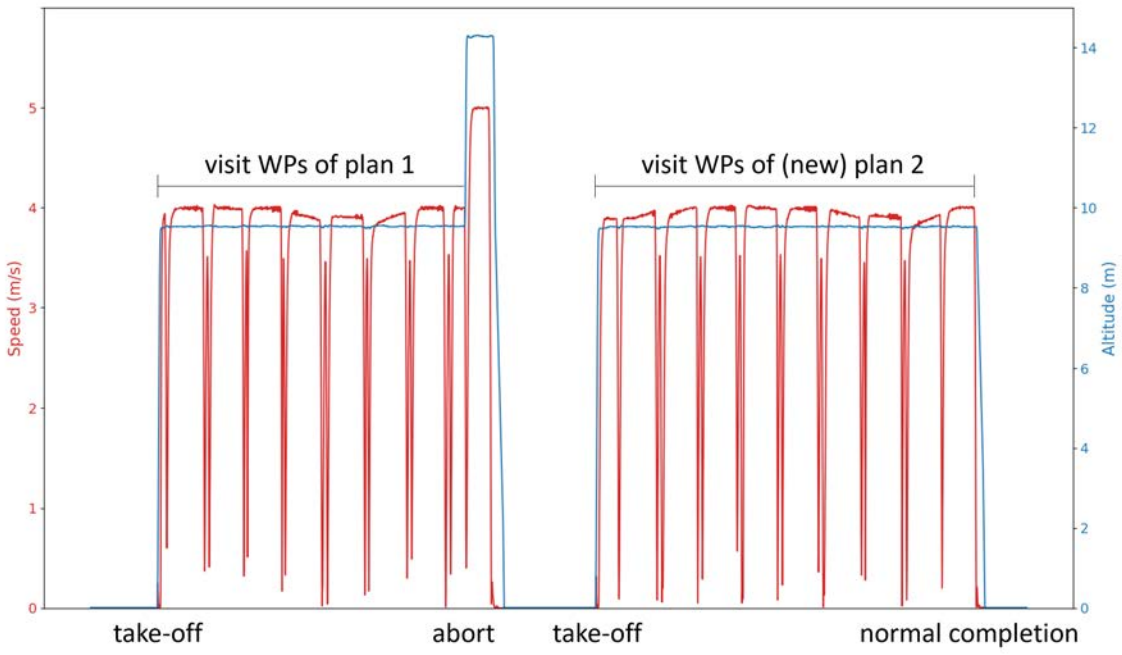
During the execution of the first plan (orange), the weather station is set to report bad conditions. As a result, the mission is aborted at the point marked in Figure 5.9b and the drone returns back to base and lands. The remaining part of the first plan is rescheduled and merged with the second plan (blue), thereby producing a new ad-hoc plan (green) that is executed as soon as the drone's batteries are fully charged.



(a) Initial offline plans.



(b) Modified second plan.



(c) Flight speed (red/left y-axis) and altitude (blue/right y-axis) of the drone.

Figure 5.9: Mission execution - One drone.

Figure 5.9c depicts the drone's speed and flight altitude relative to the home position, which are recorded during the experiment in the system logs. Once the drone takes off and starts the mission according to the first plan, its altitude remains constant at about 10 meters, while its speed is roughly 4 m/s except when the drone slows down in order to turn at the specified waypoints. When the sensing task is aborted, the mission driver program activates the return-to-land behavior. As a result, the drone immediately rises to a default altitude of 15 meters and then moves with a default speed of 5 m/s to the home position where it lands. The same basic pattern is repeated during the execution of the second plan. The difference is that, in this case, the entire mission is completed successfully, thus the drone follows the planned path back to home without changing the specified flight altitude or speed.

5.6.3 Indicative test scenario - two drones

In this example, we conduct the previous mission using two drones, Figure 5.10. The initial/home points of the drones are shown with the yellow nodes. The two plans produced for the drones are shown with the orange and blue arrows respectively.

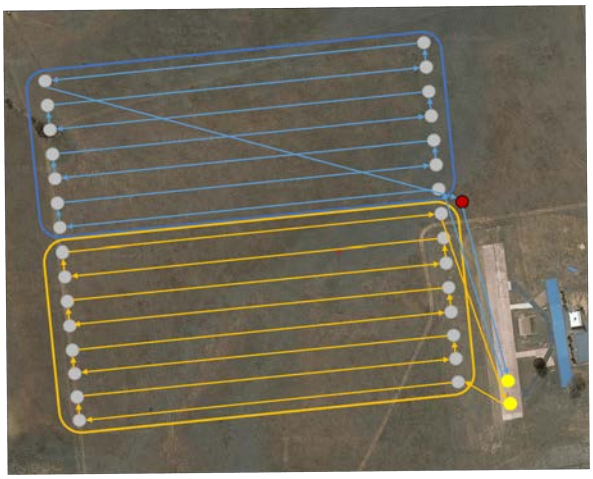


Figure 5.10: Plans generated for each drone.

Note that the drone that executes the blue plan, returns to home via a detour denoted with the red node. This intermediate node was inserted by the driver program's planner-algorithm to prevent the respective drone to enter the target area of the other one.

Like in the previous example, we record and present the speed and flight altitude relative to the home position, which are recorded during the mission, Figure 5.11. The drones takeoff at 10 meters, while this altitude is kept for the entire mission. Then with a 4 m/s speed, the drones traverse the target edges to perform the desired sensing task. When the mission ends,

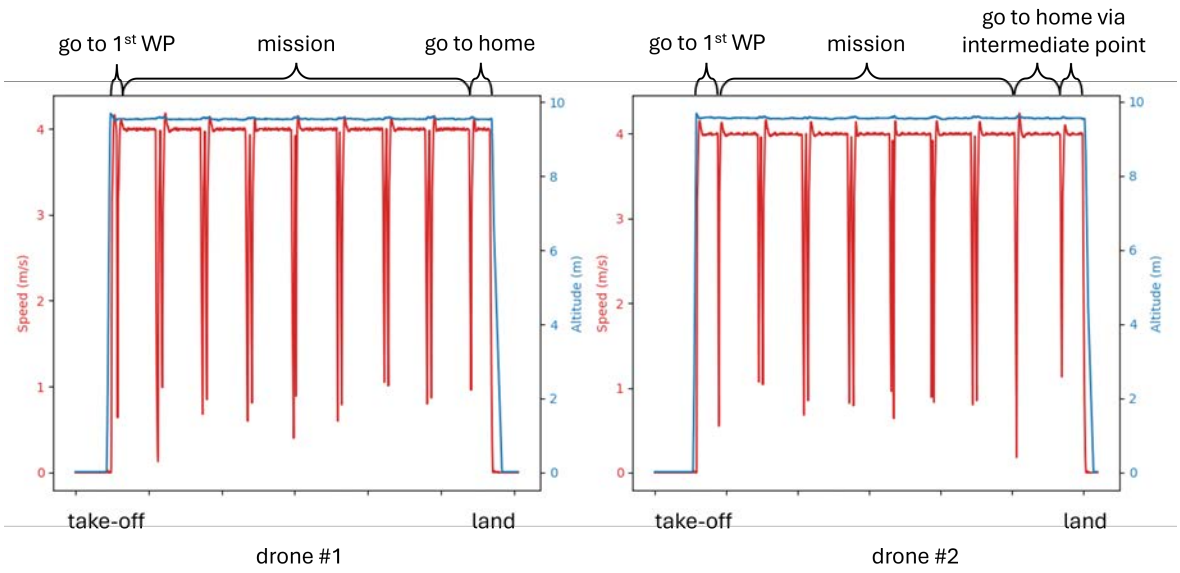


Figure 5.11: Flight speed (red/left y-axis) and altitude (blue/right y-axis) of the two drones.

the drones return to the home point and land. As explained, the second drone returns to its home point via an intermediate point.

5.6.4 Indicative test scenario - Two drones with one failure

In the last test case, we run the same experiment as the previous (shown in Figure 5.10). However, the drone following the orange plan is set to experience a failure before traversing the final edge. As a consequence, the drone aborts and is set to RTL mode, leading to the same behavior as shown in Figure 5.9c. The remaining part of the drone's mission is assigned to the second drone.

The logs for the speed and altitude recorded during this mission, are shown in Figure 5.12. As we see in the logs of drone 1, it manages to complete 7 out of 8 edges assigned to it before aborting. Then it enters in RTL mode and returns to the home point and land. When drone 2 finishes with its assigned plan, it goes to the next new assigned WP to execute the respective sensing task. After completing this new sensing task, it returns and lands.

5.6.5 Field setup

The setup used to perform tests in the field is shown in Figure 5.13. The drones we used is a custom hexacopter and a custom quadcopter (only one shown in the figure) with CUAV V5 nano autopilot boards [104], which provide all flight-related sensors and run the popular

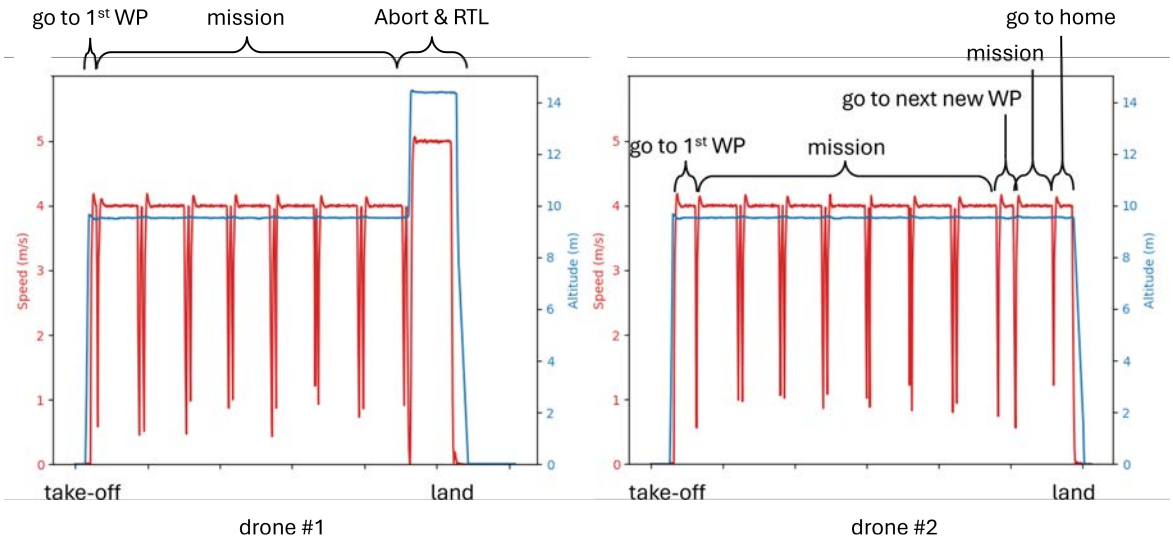


Figure 5.12: Flight speed (red y-axis) and altitude (blue y-axis) of the two drones.

ArduPilot [86] autopilot software for multicopters. The drones feature as a separate companion board a Raspberry Pi 3 Model B [128] (RPi) with a quad-core ARM Cortex A53 processor at 1.2 GHz and 1GB of RAM, running the Debian-based Raspberry Pi OS Buster, the officially supported Linux distribution for the RPi. The RPi is connected to the autopilot board over serial. The mobility service employs the DroneKit library, which communicates with the autopilot through MAVProxy [106]. The camera service accesses an onboard 8 MP RPi Camera Module v2 [129] via the picamera library [130]. In the field experiments we do not include any hangar or battery charger subsystem. Instead, the hangar service communicates with a virtual hangar that provides the same API of the real one and simulates its operation.

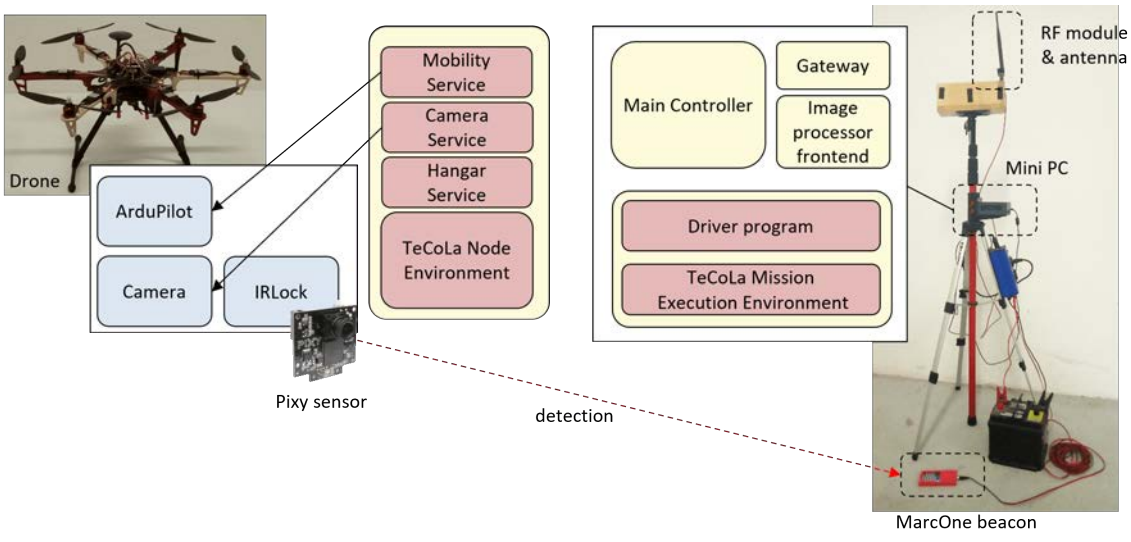


Figure 5.13: System setup used in the field tests.

For the ground station we use an Intel NUC mini PC running a Ubuntu 16.0 LTS Linux distribution, powered via an external car battery. Since the field tests focus on the operation of the core system components, this configuration does not include a weather station. Instead, the Weather API is hardwired to report good weather conditions. All tests are based on manually prepared mission plans and no rescheduling or any aborts take place in any case.

The placement of TeCoLa Node Environment can be configured. It can run directly on the companion board of the drone, or on the ground station. Additionally, the communication between the ground station and the drone is also configurable to go over WiFi or RF telemetry. All the drones and the embedded computer have the corresponding interfaces; the latter has an integrated WiFi antenna, while the RF module is externally connected to it via USB and is mounted on a pole for better coverage. The system can be configured to use only one of these interfaces, or both of them concurrently with the option of service-based binding. For instance, the RF which has a longer range can be used for the more critical mobility service, whereas the WiFi which can support higher throughput can be used for the camera service. In this setup the TeCoLa Node Environment runs on the mini PC, while the mobility service communicates with the autopilot via RF telemetry.

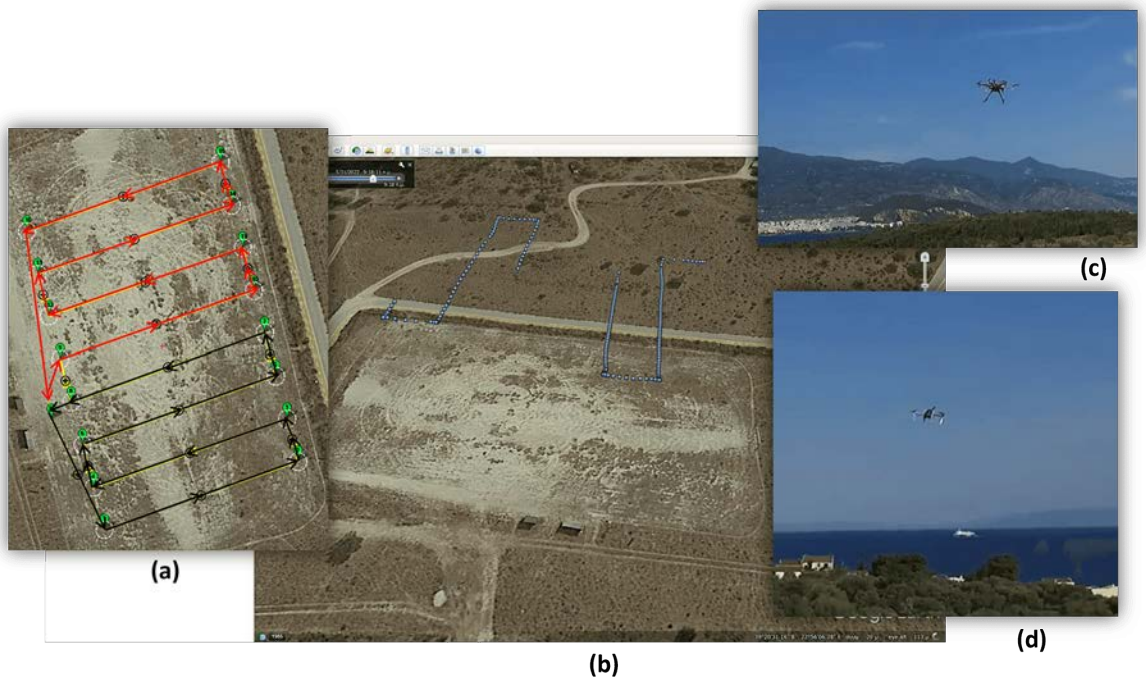


Figure 5.14: Field test with two drones engaged concurrently: (a) the two mission plans; (b) actual flight paths based on log traces; (c) & (d) snapshots of the drones mid-flight.

Figure 5.14 illustrates a field experiment with two drones. The flight plan of each drone

is given in Figure 5.14(a) while Figure 5.14(b) shows the traces of the two drones from the real logs collected during the experiment. Finally, Figure 5.14(c) and Figure 5.14(d) show snapshots of the two drones during the mission execution; given that, for safety reasons, the two drones do not fly close to each other, it was not possible to shoot a picture that captures both drones.

5.6.6 Real-world application scenario

The above system was tested for a concrete application scenario in the context of the PV-Auto-Scout project [131] with the goal to support the automated monitoring of a photovoltaic park using aerial infrared (IR) thermography. The complete system featured its own on-site weather station and (custom) hangar with precision landing support and a charging subsystem. In these field experiments, the Main Controller, Weather Observer and Hangar Service were communicating with the respective real subsystems instead a mockup.

Figure 5.15: Photovoltaic park setup.

Figure 5.15 shows pictures taken from one of the tests that were conducted in the field. In the left picture, we see the pole with the RF antenna and telemetry system connected to the embedded PC that hosts the main software of the system that controls the drone to automate the mission, along with the hangar that is used to shelter and charge the drone. In the middle picture of Figure 5.15, we can see the charging plate on the center of the hangar that was used for the charging of the drone after it completes a mission, while a MarcOne beacon [132] was used to allow the drone to land accurately on the pad. Finally, in the right picture, we see a snapshot of the drone flying over a string of panels under the control of the system.

Figure 5.16(a) shows an aerial (satellite) view of the solar park that was used for the field tests, where one can see the different strings of photovoltaic panels. Each string is a potential target path/edge to be monitored. Figure 5.16(b) illustrates a scanning pass that was performed by the drone over a string, with the thermal pictures that were taken as a result of

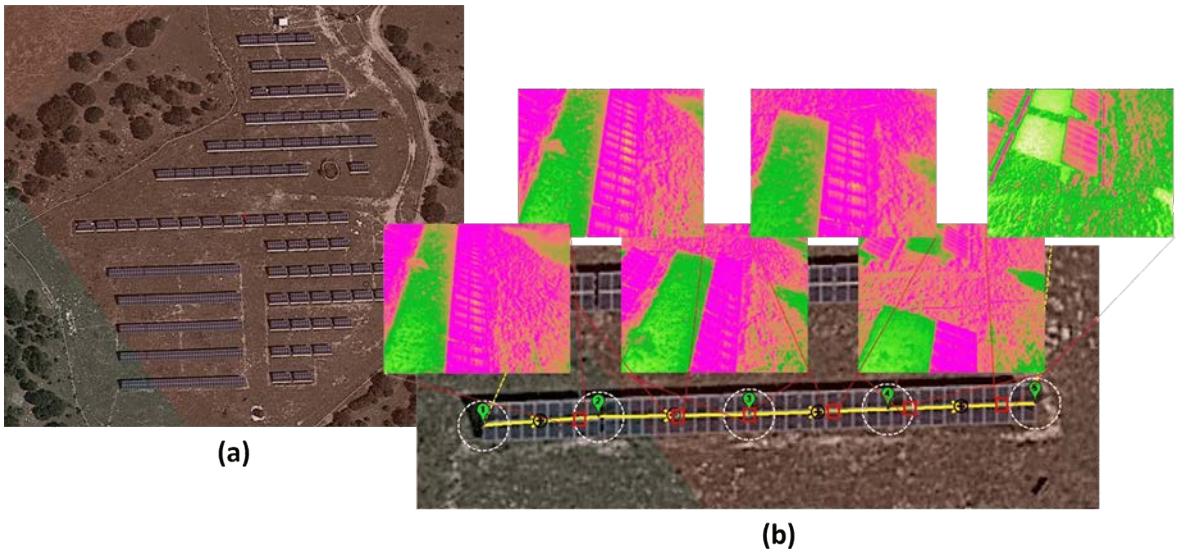


Figure 5.16: Photovoltaic park and simple mission plan.

the mission; in this particular use case, these images are downloaded and processed on the embedded PC offline (to detect potential issues of the panels, such as cracks and hotposts) once the drone completes the entire mission.

These field tests confirmed the robust operation of the system to conduct periodic as well as on-demand scanning missions, with full automation of the entire cycle of operation without any human involvement; still, a human pilot was stand-by in all field tests so that it would be possible to take over the control of the drone just in case the system malfunctioned. Successful tests were also performed using two drones, in the spirit of the simulation experiments discussed above. However, due the very significant overhead (requiring two stand-by pilots and separate landing areas), such tests were performed only for very basic mission scenarios.

Chapter 6

Conclusion

6.1 Summary

In this thesis, we have focused on applications that use drones as a key component. The general pattern of the missions require the visitation of some locations of interest, perform a sensing task, process the sensed data and depending on the sensed data perform an additional action on the spot.

The first part of our work explores different ways to minimize the mission time of such missions. We have tackled this objective in an incremental manner. To begin with, we focused on optimizing the paths of the drones by proposing an offline algorithm to build the paths and an online approach to adapt and optimize the paths on runtime when this was required. Then we turned our interest in minimizing the processing time of the sensed data so that a drone can decide faster about whether an additional action is needed on the respective point or not, reducing the hovering time of the drone. This was achieved by fairly sharing the limited resources of nearby edge servers between the drones operating in the area. For this problem we proposed different heuristics for building the schedule of each drone along with an online protocol to adapt the schedule when needed. The schedule creation consists of jointly setting the paths and the offloading decisions of each drone. To further reduce the mission time, we proposed a learning approach so that a drone can decide whether it worth waiting for a computation to complete before moving to the next point of interest or not. This decision was based on the experience of the drone from previous missions. Each of the above problems were thoroughly evaluated in a wide range of experiments.

The second part of our work investigates how to support the full automation of such drone-

based sensing applications. For this purpose, we designed a modular system architecture to automate the entire operation cycle of one or more drones. The system was tested in a wide range of scenarios in a simulated environment. The main functionality of the system was also tested in the field using two custom made drones.

6.2 Future work

The problems addressed in this thesis have shown new paths for deeper exploration and improvement. While significant progress has been made, each problem tackled presents opportunities for enhancement and further investigation. Bellow we discuss some promising directions for future research.

One interesting direction, is to explore more advanced algorithms for jointly determining the paths and offloading decisions of multiple drones. In this work, the paths are generated first, followed by an investigation of the order and direction of the different subpaths. Investigating algorithms, that enable greater exploration of potential paths could yield good results. Additionally, when drones have heterogeneous onboard processing units, an intriguing extension of the problem involves enabling drones to offload computations to one another. In this scenario, drones act as both clients and servers. This brings to the problem additional complexity given that the drones may follow entirely different paths while having limited communication ranges.

For our work that focuses on a drone that learns from past experience about whether it should wait or not for a computation to complete, there are plenty of interesting directions. A possible extension to this work is to investigate a wider space of control decisions. For instance, when deciding to go, the drone could also set the target flying speed depending on the likelihood of event detection at the previous point of interest, choosing higher speeds when event detection is highly unlikely and lower speeds otherwise to minimize the penalty in case the decision to go proves to be wrong. Another direction is to investigate swarm scenarios, where the area of interest is scanned by multiple drones. In this case, when a drone wrongfully decides to go may be aided by another drone that is closer to the respective point, or even redistribute the points of interest, so that the drone that was delayed due to the wrong decision “offloads” some of its tasks to another drone.

Finally, looking at the autonomous drone system of the final chapter, the most intuitive

future direction is to implement different driver programs to tackle more complex missions. One such driver program could invoke the Image Processor subsystem on the flight to perform the required data processing, in case the mission is data-driven. Additionally, the drone environment could be extended with an additional service for the processing tasks. This service could abstract the complexity to flexibly decide whether the task should be offloaded to the ground station or processed on the onboard computer of the respective drone.

References

[1] Business research. https://www.businessresearchinsights.com/market-reports/civilian-drone-market-100513.

[2] DroneKit. https://dronekit.io/.

[3] K Jayalath and SR Munasinghe. Drone-based autonomous human identification for search and rescue missions in real-time. In *2021 10th International Conference on Information and Automation for Sustainability (ICIAfS)*, pages 518–523, 2021.

[4] Anastasiia Rozhok, Emanuele Adorni, and Roberto Revetria. Uavs as means to provide first aid kits to lost at sea victims, 2022.

[5] Manish Anand Pawar, Tanmay Liladhar Talele, Sanket Dilip Vagal, and Tatwadarshi P Nagarhalli. Farmatron-pest detection and treatment using ai based drone. *Int. J. Res. Appl. Sci. Eng. Technol*, 8:19–24, 2020.

[6] Uma Meleti, Abolfazl Razi, and Fatemeh Afghah. Obscured wildfire flame detection by spatio-temporal analysis of smoke patterns using frame-wise transformers. In *2024 20th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, pages 58–65, 2024.

[7] Sourav Barua, Meer Shadman Shafkat Tanjim, Ashrafun Nushra Oishi, Sudeb Chandra Das, Md Abul Basar, and Shoyeb Ahammad Rafi. Design and implementation of fire extinguishing ball thrower quadcopter. In *2020 IEEE region 10 symposium (TENSYMP)*, pages 1404–1407, 2020.

[8] Giorgos Polychronis and Spyros Lalis. Tournament selection algorithm for the multiple travelling salesman problem. In *6th International Conference on Vehicle Technology and Intelligent Transport Systems, (VEHITS)*, pages 585–594, 2020.

[9] Giorgos Polychronis and Spyros Lalis. Dynamic multiple vehicle routing under energy capacity constraints. In *IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, pages 1–7, 2020.

[10] Giorgos Polychronis and Spyros Lalis. Safe optimistic path planning for autonomous drones under dynamic energy costs. In *IEEE International Intelligent Transportation Systems Conference (ITSC)*, pages 1927–1933, 2021.

[11] Giorgos Polychronis and Spyros Lalis. Planning computation offloading on shared edge infrastructure for multiple drones. In *IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 302–307, 2022.

[12] Giorgos Polychronis and Spyros Lalis. Joint edge resource allocation and path planning for drones with energy constraints. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 378–399, 2022.

[13] Giorgos Polychronis and Spyros Lalis. Flexible computation offloading at the edge for autonomous drones with uncertain flight times. In *19th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, pages 201–208, 2023.

[14] Giorgos Polychronis and Spyros Lalis. Scheduling offloading decisions for heterogeneous drones on shared edge resources. In *Device-Edge-Cloud Continuum: Paradigms, Architectures and Applications*, pages 69–88. Springer, 2023.

[15] Giorgos Polychronis, Manos Koutsoubelias, and Spyros Lalis. Should i stay or should i go: A learning approach for drone-based sensing applications. In *20th International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, pages 339–346, 2024.

[16] Manos Koutsoubelias, Nasos Grigoropoulos, Giorgos Polychronis, Giannis Badakis, and Spyros Lalis. System architecture for autonomous drone-based remote sensing. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 220–242, 2021.

[17] SP Anbuudayasankar, K Ganesh, and Sanjay Mohapatra. Survey of methodologies for tsp and vrp. In *Models for Practical Routing Problems in Logistics*, pages 11–42. Springer, 2014.

[18] Gregory Gutin and Abraham P Punnen. *The traveling salesman problem and its variations*, volume 12. Springer Science & Business Media, 2006.

[19] Donald Davendra. *Traveling Salesman Problem: Theory and Applications*. BoD–Books on Demand, 2010.

[20] Arthur E Carter and Cliff T Ragsdale. A new approach to solving the multiple traveling salesperson problem using genetic algorithms. *European journal of operational research*, 175(1):246–257, 2006.

[21] Shuai Yuan, Bradley Skinner, Shoudong Huang, and Dikai Liu. A new crossover approach for solving the multiple travelling salesmen problem using genetic algorithms. *European Journal of Operational Research*, 228(1):72–82, 2013.

[22] Emanuel Falkenauer. The grouping genetic algorithms-widening the scope of the gas. *Belgian Journal of Operations Research, Statistics and Computer Science*, 33(1):2, 1992.

[23] Evelyn C Brown, Cliff T Ragsdale, and Arthur E Carter. A grouping genetic algorithm for the multiple traveling salesperson problem. *International Journal of Information Technology & Decision Making*, 6(02):333–347, 2007.

[24] Alok Singh and Anurag Singh Baghel. A new grouping genetic algorithm approach to the multiple traveling salesperson problem. *Soft Computing*, 13(1):95–101, 2009.

[25] Pandiri Venkatesh and Alok Singh. Two metaheuristic approaches for the multiple traveling salesperson problem. *Applied Soft Computing*, 26:74–89, 2015.

[26] Weimin Liu, Sujian Li, Fanggeng Zhao, and Aiyun Zheng. An ant colony optimization algorithm for the multiple traveling salesmen problem. In *2009 4th IEEE conference on industrial electronics and applications*, pages 1533–1537, 2009.

[27] Ilari Vallivaara. A team ant colony optimization algorithm for the multiple travelling salesmen problem with minmax objective. In *Proceedings of the 27th IASTED International Conference on Modelling, Identification and Control*, pages 387–392, 2008.

[28] Samerkae Somhom, Abdolhamid Modares, and Takao Enkawa. Competition-based neural network for the multiple travelling salesmen problem with minmax objective. *Computers & Operations Research*, 26(4):395–407, 1999.

[29] Vlad-Ioan Lupoaie, Ivona-Alexandra Chili, Mihaela Elena Breaban, and Madalina Raschip. Som-guided evolutionary search for solving minmax multiple-tsp. In *2019 IEEE Congress on Evolutionary Computation (CEC)*, pages 73–80, 2019.

[30] Yongzhen Wang, Yan Chen, and Yan Lin. Memetic algorithm based on sequential variable neighborhood descent for the minmax multiple traveling salesman problem. *Computers & Industrial Engineering*, 106:105–122, 2017.

[31] Banu Soylu. A general variable neighborhood search heuristic for multiple traveling salesmen problem. *Computers & Industrial Engineering*, 90:390–401, 2015.

[32] Paulo M França, Michel Gendreau, Gilbert Laporte, and Felipe M Müller. The m-traveling salesman problem with minmax objective. *Transportation Science*, 29(3):267–275, 1995.

[33] Bruce L Golden, Gilbert Laporte, and Éric D Taillard. An adaptive memory heuristic for a class of vehicle routing problems with minmax objective. *Computers & Operations Research*, 24(5):445–452, 1997.

[34] Isaac Vandermeulen, Roderich Groß, and Andreas Kolling. Balanced task allocation by partitioning the multiple traveling salesperson problem. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1479–1487, 2019.

[35] Luca Bertazzi, Bruce Golden, and Xingyin Wang. Min–max vs. min–sum vehicle routing: A worst-case analysis. *European Journal of Operational Research*, 240(2):372–381, 2015.

[36] Ulrike Ritzinger, Jakob Puchinger, and Richard F Hartl. A survey on dynamic and stochastic vehicle routing problems. *International Journal of Production Research*, 54(1):215–231, 2016.

[37] Harilaos N Psaraftis, Min Wen, and Christos A Kontovas. Dynamic vehicle routing problems: Three decades and counting. *Networks*, 67(1):3–31, 2016.

[38] Victor Pillac, Michel Gendreau, Christelle Guéret, and Andrés L Medaglia. A review of dynamic vehicle routing problems. *European Journal of Operational Research*, 225(1):1–11, 2013.

[39] Huey-Kuo Chen, Che-Fu Hsueh, and Mei-Shiang Chang. The real-time time-dependent vehicle routing problem. *Transportation Research Part E: Logistics and Transportation Review*, 42(5):383–408, 2006.

[40] Ilhan Or. Traveling salesman-type combinatorial problems and their relation to the logistics of blood banking. *PhD thesis (Department of Industrial Engineering and Management Science, Northwestern University)*, 1976.

[41] Ali Haghani and Soojung Jung. A dynamic vehicle routing problem with time-dependent travel times. *Computers & Operations Research*, 32(11):2959–2986, 2005.

[42] Eiichi Taniguchi and Hiroshi Shimamoto. Intelligent transportation system based dynamic vehicle routing and scheduling with variable travel times. *Transportation Research Part C: Emerging Technologies*, 12(3-4):235–250, 2004.

[43] Jean-Yves Potvin, Ying Xu, and Ilham Benyahia. Vehicle routing and scheduling with dynamic travel times. *Computers & Operations Research*, 33(4):1129–1137, 2006.

[44] Sandro Lorini, Jean-Yves Potvin, and Nicolas Zufferey. Online vehicle routing and scheduling with dynamic travel times. *Computers & Operations Research*, 38(7):1086–1090, 2011.

[45] Éric Taillard, Philippe Badeau, Michel Gendreau, François Guertin, and Jean-Yves Potvin. A tabu search heuristic for the vehicle routing problem with soft time windows. *Transportation Science*, 31(2):170–186, 1997.

[46] Jean Respen, Nicolas Zufferey, and Jean-Yves Potvin. Online vehicle routing and scheduling with continuous vehicle tracking. 2014.

[47] Michael Schilde, Karl F Doerner, and Richard F Hartl. Integrating stochastic time-dependent travel speed in solution methods for the dynamic dial-a-ride problem. *European Journal of Operational Research*, 238(1):18–30, 2014.

[48] Zhihai Xiang, Chengbin Chu, and Haoxun Chen. The study of a dynamic dial-a-ride problem under time-dependent and stochastic environments. *European Journal of Operational Research*, 185(2):534–551, 2008.

[49] Daniela Rojas Viloria, Elyn L Solano-Charris, Andrés Muñoz-Villamizar, and Jairo R Montoya-Torres. Unmanned aerial vehicles/drones in vehicle routing problems: a literature review. *Intl Transactions in Operational Research*, 28:1626–1657, 2021.

[50] Gino J Lim, Seonjin Kim, Jaeyoung Cho, Yibin Gong, and Amin Khodaei. Multi-uav pre-positioning and routing for power network damage assessment. *IEEE Transactions on Smart Grid*, 9(4):3643–3651, 2016.

[51] Darshan Chauhan, Avinash Unnikrishnan, and Miguel Figliozzi. Maximum coverage capacitated facility location problem with range constrained drones. *Transportation Research Part C: Emerging Technologies*, 99:1–18, 2019.

[52] Cristian Ramirez Atencia, Javier Del Ser, and David Camacho. Weighted strategies to guide a multi-objective evolutionary algorithm for multi-uav mission planning. *Swarm and Evolutionary Computation*, 44:480–495, 2019.

[53] Boualem Rabta, Christian Wankmüller, and Gerald Reiner. A drone fleet model for last-mile distribution in disaster relief operations. *Intl Journal of Disaster Risk Reduction*, 28:107–112, 2018.

[54] Kevin Dorling, Jordan Heinrichs, Geoffrey G Messier, and Sebastian Magierowski. Vehicle routing problems for drone delivery. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 47(1):70–85, 2016.

[55] Chun Cheng, Yossiri Adulyasak, and Louis-Martin Rousseau. *Formulations and exact algorithms for drone routing problem*. CIRRELT Tech. Report, 2018.

[56] Jalil Modares, Farshad Ghanei, Nicholas Mastronarde, and Karthik Dantu. Ub-anc planner: Energy efficient coverage path planning with multiple drones. In *IEEE Intl Conf. on Robotics and Automation (ICRA)*, pages 6182–6189, 2017.

[57] Joseph YJ Chow. Dynamic uav-based traffic monitoring under uncertainty as a stochastic arc-inventory routing policy. *Intl Journal of transportation science and technology*, 5(3):167–185, 2016.

[58] Dimitrios Zorbas, Luigi Di Puglia Pugliese, Tahiry Razafindralambo, and Francesca Guerriero. Optimal drone placement and cost-efficient target coverage. *Journal of Network and Computer Applications*, 75:16–31, 2016.

[59] Yanchao Liu. An optimization-driven dynamic vehicle routing algorithm for on-demand meal delivery using drones. *Computers & Operations Research*, 111:1–20, 2019.

[60] Bruno N Coelho, Vitor N Coelho, Igor M Coelho, Luiz S Ochi, Roozbeh Haghnazar, Demetrius Zuidema, Milton SF Lima, and Adilson R da Costa. A multi-objective green uav routing problem. *Computers & Operations Research*, 88:306–315, 2017.

[61] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *International conference on principles and practice of constraint programming*, pages 417–431, 1998.

[62] Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4):376–384, 1991.

[63] Cheng Zhan, Han Hu, Xiufeng Sui, Zhi Liu, Jianan Wang, and Honggang Wang. Joint resource allocation and 3d aerial trajectory design for video streaming in uav communication systems. *IEEE Transactions on Circuits and Systems for Video Technology*, 31(8):3227–3241, 2020.

[64] Jingyu Xiong, Hongzhi Guo, and Jiajia Liu. Task offloading in uav-aided edge computing: Bit allocation and trajectory optimization. *IEEE Communications Letters*, 23(3):538–541, 2019.

[65] Jiao Zhang, Li Zhou, Qi Tang, Edith C-H Ngai, Xiping Hu, Haitao Zhao, and Jibo Wei. Stochastic computation offloading and trajectory scheduling for uav-assisted mobile edge computing. *IEEE Internet of Things Journal*, 6(2):3688–3699, 2018.

[66] Ahmed Al-Hilo, Moataz Samir, Chadi Assi, Sanaa Sharafeddine, and Dariush Ebrahimi. Uav-assisted content delivery in intelligent transportation systems-joint trajectory planning and cache management. *IEEE Transactions on Intelligent Transportation Systems*, 22(8):5155–5167, 2020.

[67] Muhammad Asim, Wali Khan Mashwani, Samir Brahim Belhaouari, and Saima Hassan. A novel genetic trajectory planning algorithm with variable population size for multi-uav-assisted mobile edge computing system. *IEEE Access*, 9:125569–125579, 2021.

[68] Theodoros Kasidakis, Giorgos Polychronis, Manos Koutsoubelias, and Spyros Lalis. Reducing the mission time of drone applications through location-aware edge computing. In *Proc. IEEE International Conference on Fog and Edge Computing (ICFEC)*, pages 45–52, 2021.

[69] Bongjae Kim, Hong Min, Junyoung Heo, and Jinman Jung. Dynamic computation offloading scheme for drone-based surveillance systems. *Sensors*, 18(9):2982, 2018.

[70] Aaron Paulos, Soura Dasgupta, Jacob Beal, Yuanqiu Mo, Jon Schewe, Alexander Wald, Partha Pal, Richard Schantz, and J Bryan Lyles. Priority-enabled load balancing for dispersed computing. In *Proc. IEEE International Conference on Fog and Edge Computing (ICFEC)*, pages 1–8, 2021.

[71] Georgios Stavrinides and Helen Karatza. A hybrid approach to scheduling real-time iot workflows in fog and cloud environments. *Multimedia Tools and Applications*, 78(17):24639–24655, 2019.

[72] Georgios Stavrinides and Helen Karatza. Orchestrating real-time iot workflows in a fog computing environment utilizing partial computations with end-to-end error propagation. *Cluster Computing*, 24(4):3629–3650, 2021.

[73] Ioanna Stypsanelli, Olivier Brun, and Balakrishna J Prabhu. Performance evaluation of some adaptive task allocation algorithms for fog networks. In *Proc. IEEE International Conference on Fog and Edge Computing (ICFEC)*, pages 84–88, 2021.

[74] Georgios Stavrinides and Helen Karatza. Orchestrating bag-of-tasks applications with dynamically spawned tasks in a distributed environment. In *Proc. International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 1–8, 2021.

[75] Dmitrii Chemodanov, Chengyi Qu, Osunkoya Opeoluwa, Songjie Wang, and Prasad Calyam. Policy-based function-centric computation offloading for real-time drone video

analytics. In *IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–6, 2019.

[76] Jienan Chen, Siyu Chen, Siyu Luo, Qi Wang, Bin Cao, and Xiaoqian Li. An intelligent task offloading algorithm (itoa) for uav edge computing network. *Digital Communications and Networks*, 6(4):433–443, 2020.

[77] Xuan-Qui Pham, Nguyen Doan Man, Nguyen Dao Tan Tri, Ngo Quang Thai, and Eui-Nam Huh. A cost-and performance-effective approach for task scheduling based on collaboration between cloud and fog computing. *International Journal of Distributed Sensor Networks*, 13(11):1550147717742073, 2017.

[78] Wenda Tang, Xuan Zhao, Wajid Rafique, Lianyong Qi, Wanchun Dou, and Qiang Ni. An offloading method using decentralized p2p-enabled mobile edge servers in edge computing. *Journal of Systems Architecture*, 94:1–13, 2019.

[79] Chongwu Dong and Wushao Wen. Joint optimization for task offloading in edge computing: An evolutionary game approach. *Sensors*, 19(3):740, 2019.

[80] Qinglan Peng, Chunrong Wu, Yunni Xia, Yong Ma, Xu Wang, and Ning Jiang. Dosra: A decentralized approach to online edge task scheduling and resource allocation. *IEEE Internet of Things Journal*, 9(6):4677–4692, 2021.

[81] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100, 1997.

[82] George F. Riley and Thomas R. Henderson. The ns-3 network simulator. In *Modeling and Tools for Network Simulation*, pages 15–34. Springer, 2010.

[83] Joseph Redmon and Ali Farhadi. Yolo v3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[84] Dirk Merkel et al. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239), 2014.

[85] SITL Simulator software in the loop. https://ardupilot.org/dev/docs/sitl-simulator-software-in-the-loop.html.

[86] Ardupilot. https://ardupilot.org.

[87] Mohamed Ayoub Messous, Hermann Hellwagner, Sidi-Mohammed Senouci, Driton Emini, and Dominik Schnieders. Edge computing for visual navigation and mapping in a uav network. In *ICC IEEE International Conference on Communications (ICC)*, pages 1–6, 2020.

[88] Akhil Bandarupalli, Sarthak Jain, Akash Melachuri, Joseph Pappas, and Somali Chaterji. Vega: Drone-based multi-altitude target detection for autonomous surveillance. In *IEEE International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT)*, pages 209–216, 2023.

[89] Gopi Gugan and Anwar Haque. Path planning for autonomous drones: Challenges and future directions. *Drones*, 7(3):169, 2023.

[90] Ilker Kucukoglu, Reginald Dewil, and Dirk Cattrysse. The electric vehicle routing problem and its variations: A literature review. *Computers & Industrial Engineering*, 161:107650, 2021.

[91] Muhammad Asim, Wali Khan Mashwani, Habib Shah, and Samir Brahim Belhaouari. An evolutionary trajectory planning algorithm for multi-uav-assisted mec system. *Soft Computing*, 26(16):7479–7492, 2022.

[92] Jianqing Lin and Linqiang Pan. Multiobjective trajectory optimization with a cutting and padding encoding strategy for single-uav-assisted mobile edge computing system. *Swarm and Evolutionary Computation*, 75:101163, 2022.

[93] Christian H Christiansen and Jens Lysgaard. A branch-and-price algorithm for the capacitated vehicle routing problem with stochastic demands. *Operations Research Letters*, 35(6):773–781, 2007.

[94] Justin C Goodson, Jeffrey W Ohlmann, and Barrett W Thomas. Cyclic-order neighborhoods with application to the vehicle routing problem with stochastic demand. *European Journal of Operational Research*, 217(2):312–323, 2012.

[95] Sergio Gonzalez-Martin, Angel A Juan, Daniel Riera, Monica G Elizondo, and Juan J Ramos. A simheuristic algorithm for solving the arc routing problem with stochastic demands. *Journal of Simulation*, 12(1):53–66, 2018.

[96] Waheed Iqbal, Matthew N Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6):871–879, 2011.

[97] Young Choon Lee and Albert Y Zomaya. Energy efficient utilization of resources in cloud computing systems. *The Journal of Supercomputing*, 60(2):268–280, 2012.

[98] Chaima Ghribi, Makhlouf Hadji, and Djamal Zeghlache. Energy efficient vm scheduling for cloud data centers: Exact allocation and migration algorithms. In *13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGRID)*, pages 671—678, 2013.

[99] Rahul Yadav, Weizhe Zhang, Omprakash Kaiwartya, Prabhat Ranjan Singh, Ibrahim A Elgendy, and Yu-Chu Tian. Adaptive energy-aware algorithms for minimizing energy consumption and sla violation in cloud computing. *IEEE Access*, 6, 2018.

[100] Douglas C Montgomery, Elizabeth A Peck, and G Geoffrey Vining. *Introduction to linear regression analysis*. John Wiley & Sons, 2021.

[101] Wei-Yin Loh. Classification and regression trees. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 1(1):14–23, 2011.

[102] David JC MacKay. Bayesian interpolation. *Neural computation*, 4(3):415–447, 1992.

[103] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. Lof: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, pages 93–104, 2000.

[104] Cuav. https://www.cuav.net/en/v5-nano-2/.

[105] Mavlink. https://mavlink.io/en/.

[106] Mavproxy. https://ardupilot.org/mavproxy/.

[107] Yolo. https://yolov8.com/.

[108] Paweł Smyczyński, Łukasz Starzec, and Grzegorz Granosik. Autonomous drone control system for object tracking: Flexible system design with implementation example. In *IEEE International Conference on Methods and Models in Automation and Robotics (MMAR)*, pages 734–738, 2017.

[109] Michail Kalaitzakis, Sreehari Rajan Kattil, Nikolaos Vitzilaios, Dimitris Rizos, and Michael Sutton. Dynamic structural health monitoring using a dic-enabled drone. In *International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 321–327, 2019.

[110] N Souli, R Makrigiorgis, A Anastasiou, A Zacharia, P Petrides, A Lazanas, P Valianti, P Kolios, and G Ellinas. Horizonblock: Implementation of an autonomous counter-drone system. In *International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 398–404, 2020.

[111] Jayson Boubin, John Chumley, Christopher Stewart, and Sami Khanal. Autonomic computing challenges in fully autonomous precision agriculture. In *IEEE International Conference on Autonomic Computing (ICAC)*, pages 11–17, 2019.

[112] Manos Koutsoubelias and Spyros Lalis. Tecola: A programming framework for dynamic and heterogeneous robotic teams. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous)*, pages 115—124, 2016.

[113] Martin Carroll, Kedar S Namjoshi, and Itai Segall. The resh programming language for multirobot orchestration. *arXiv preprint arXiv:2103.13921*, 2021.

[114] Keila Lima, Eduardo RB Marques, José Pinto, and Joao B Sousa. Dolphin: a task orchestration language for autonomous vehicle networks. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 603–610, 2018.

[115] Roberto Casadei, Gianluca Aguzzi, and Mirko Viroli. A programming approach to collective autonomy. *Journal of Sensor and Actuator Networks*, 10(2):27, 2021.

[116] Oliver Kosak, Lukas Huhn, Felix Bohn, Constantin Wanninger, Alwin Hoffmann, and Wolfgang Reif. Maple-swarm: programming collective behavior for ensembles by extending htn-planning. In *International Symposium on Leveraging Applications of Formal Methods*, pages 507–524, 2020.

[117] Justin Yapp, Remzi Seker, and Radu Babiceanu. Uav as a service: Enabling on-demand access and on-the-fly re-tasking of multi-tenant uavs using cloud services. In *IEEE/AIAA Digital Avionics Systems Conference (DASC)*, pages 1–8, 2016.

[118] Gabriele Ermacora, Stefano Rosa, and Antonio Toma. Fly4smartcity: A cloud robotics service for smart city applications. *Journal of Ambient Intelligence and Smart Environments*, 8(3):347–358, 2016.

[119] Anis Koubâa, Basit Qureshi, Mohamed-Foued Sriti, Azza Allouch, Yasir Javed, Maram Alajlan, Omar Cheikhrouhou, Mohamed Khalgui, and Eduardo Tovar. Dronemap planner: A service-oriented cloud-based management system for the internet-of-drones. *Ad Hoc Networks*, 86:46–62, 2019.

[120] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[121] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering (FOSE)*, pages 259–268, 2007.

[122] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Florian Dudouet, and Andrew Edmonds. An architecture for self-managing microservices. In *International Workshop on Automated Incident Management in Cloud*, pages 19–24, 2015.

[123] Sukhpal Singh Gill, Inderveer Chana, Maninder Singh, and Rajkumar Buyya. Radar: Self-configuring and self-healing in resource management for enhancing quality of cloud services. *Concurrency and Computation: Practice and Experience*, 31(1):e4834, 2019.

[124] Ana Juan Ferrer, Soeren Becker, Florian Schmidt, Lauritz Thamsen, and Odej Kao. Towards a cognitive compute continuum: An architecture for ad-hoc self-managed swarms. *arXiv preprint arXiv:2103.06026*, 2021.

[125] Geon-Hwan Kim, Jae-Choong Nam, Imtiaz Mahmud, and You-Ze Cho. Multi-drone control and network self-recovery for flying ad hoc networks. In *International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 148–150, 2016.

[126] Ting Yang, Chuan Heng Foh, Fabien Heliot, Chee Yen Leow, and Periklis Chatzimisios. Self-organization drone-based unmanned aerial vehicles (uav) networks. In *IEEE International Conference on Communications (ICC)*, pages 1–6, 2019.

[127] Angel Corberán, Richard Eglese, Geir Hasle, Isaac Plana, and José María Sanchis. Arc routing problems: A review of the past, present, and future. *Networks*, 77(1):88–115, 2021.

[128] Raspberry Pi 3. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[129] Raspberry Pi camera. https://www.raspberrypi.org/products/camera-module-v2/.

[130] Picamera interface. https://github.com/waveform80/picamera.

[131] PV-Auto-Scout. https://csl.e-ce.uth.gr/projects/pv-auto-scout/.

[132] Markone beacon. https://irlock.com/products/markone-v1-1.