SOFTWARE APPROACHES TO OPTIMIZE ENERGY CONSUMPTION FOR A TEAM OF DISTRIBUTED AUTONOMOUS MOBILE ROBOTS

SOFTWARE APPROACHES TO OPTIMIZE ENERGY CONSUMPTION FOR A TEAM OF DISTRIBUTED AUTONOMOUS MOBILE ROBOTS

BY

ANH-DUY VU M.Eng.

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

Doctor of Philosophy (2019) McMaster University

(Computing and Software) Hamilton, Ontario, Canada

TITLE: Software Approaches to Optimize Energy Consumption for a Team of Distributed Autonomous Mobile Robots

AUTHOR: Anh-Duy Vu

M.Eng. (Software Engineering),

Kookmin University, Seoul, South Korea

SUPERVISOR: George Karakostas

Abstract

In recent years, we have seen the applications of *distributed autonomous mobile robots (DAMRs)* in a broad spectrum of areas like search and rescue, disaster management, warehouse, and delivery systems. Although each type of systems employing DAMRs has its specific challenges, they are all limited by energy since the robots are powered by batteries which have not advanced in decades. This motivates the development of energy efficiency for such systems.

Although there has been research on optimizing energy for robotic systems, their approaches are from low-level (e.g., mechanic, system control, or avionic) perspectives. They, therefore, are limited to a specific type of robots and not easily adjusted to apply for different types of robots. Moreover, there is a lack of work studying the problem from a software perspective and abstraction.

In this thesis, we tackle the problem from a software perspective and are particularly interested in DAMR systems in which a team of networked robots navigating in a physical environment and acting in concert to accomplish a common goal. Also, the primary focus of our work is to design schedules (or plans) for the robots so that they can achieve their goal while spending as little energy as possible. To this end, we study the problem in three different contexts:

- *Managing reliability and energy consumption tradeoff.* That is, we propose that robots *verify* computational results of one another to increase the corroboration of outputs of our DAMR systems. However, this new feature requires robots to do additional tasks and consume more energy. Thus, we propose approaches to reach a *balance* between *energy consumption* and the *reliability* of results obtained by our DAMR systems.

- *Extending the operational time of robots.* We first propose that our DAMR systems should employ *charging stations* where robots can come to recharge their batteries. Then, we aim to design schedules for the robots so that they can finish all their tasks while consuming as little energy and time (including the time spent for recharging) as possible. Moreover, we model the working space by a *connected (possibly incomplete)* graph to make the problem more practical.

- *Coping with environmental changes.* This path planning problem takes into account not only energy limits but also *changes in the physical environment*, which may result in overheads (i.e., additional time and energy) that robots incur while doing their tasks. To tackle the problem from a software perspective, we first utilize Gaussian Process and Polynomial Regression to model disturbances and energy consumption, respectively, then proposed techniques to generate plans and adjust them when robots detect environmental changes.

For each problem, we give a formal description, a *transformation to integer (linear) programming*, *offline algorithms*, and an *online algorithm*. Moreover, we also rigorously analyze the proposed techniques by conducting simulations and *experiments* in a real network of *unmanned aerial vehicles (UAVs)*.

To my parents, Ngoc-Thuy Vu and Thi-Thanh Nguyen, my beautiful wife, Thuy-Dung Nguyen, and my lovely daughters, Denysia and Diona Vu, for their unconditional love, encouragement, and support.

Acknowledgments

First of all, I thank my supervisor, Professor George Karakostas, for his continuous support of my Ph.D. study and for his immense knowledge. His guidance helped me with my research and the writing of this thesis. I would also like thank to Professor Borzoo Bonakdarpour for offering me technical, and moral support.

Also, I would like to truly thank the Department of Computing and Software at McMaster University for offering me financial support through teaching assistantships during the four years of my Ph.D. program.

I would like to take this opportunity to thank my colleagues Ian McAuthur, Noel Bret, Shokoufeh Kazemlou, Akhil Krishnan, Mikhail Markov, and Umair Siddique; because of them my study here has been very enjoyable and memorable.

I cannot express enough gratitude to my parents, Ngoc-Thuy and Thi-Thanh, for their continuous support every step of the way. They have pushed me forward at every juncture of my life, and without them I would not be writing a PhD thesis acknowledgment.

Finally, I would like to thank my wife, Thuy-Dung. Thank you for your support, encouragement, and belief. I would also like to thank our daughters, Denysia and Diona, for making my life brighter.

Contents

List of Figures

Abbreviations

CPS	Cyber-physical Systems
DAG	Directed Acyclic Graph
DAMR	Distributed Autonomous Mobile Robot
GPS	Global Positioning System
IoT	Internet of Things
ILP	Integer Linear Programming
MAC	Message Authentication Code
MILP	Multiobjective Integer Linear Programming
MIP	Multiobjective Integer Programming
mTSP	Multiple Traveling Salesman Problem
NNH	Nearest Neighbor Heuristic
OPPD	Optimal Path Planning in the Presence of Disturbances
OR	Optimal Recharging

RET	Reliability-Energy Tradeoff
RRT	Rapidly-exploring Random Tree
STSP	Steiner Traveling Salesman Problem
TSP	Traveling Salesman Problem
UAV	Unmanned Aerial Vehicle
VRP	Vehicles Routing Problem

Chapter 1

Introduction

Although *distributed autonomous mobile robotics* is a young field, it offers the possibility of fundamentally changing a myriad of areas like exploration (Burgard *et al.*, 2000; Rekleitis *et al.*, 2001; Simmons *et al.*, 2000; Burgard *et al.*, 2005), surveillance (Kolling and Carpin, 2008, 2006; Parker, 1999; Ahmadi and Stone, 2006), search and rescue (Baxter *et al.*, 2007; Calisi *et al.*, 2007; Macwan *et al.*, 2015), transportation (Donald *et al.*, 2000; Rus *et al.*, 1995; Stilwell and Bay, 1993; Wang *et al.*, 2000; Khatib *et al.*, 1996; Sugar and Kumar, 2000), and delivery systems (DHL International GmbH, 2016; McFarland, 2016; Amazon.com, Inc., 2016; Gatteschi *et al.*, 2015; Wing LLC, 2019). The most remarkable successes can be seen in the case of autonomous vehicles (particularly in self-driving cars) where automobile manufacturers like Tesla (Tesla, Inc., 2008), transportation network companies like Uber (Uber Technologies, Inc., 2009), technology companies like Waymo (Waymo LLC, 2009) (formerly the Google self-driving car project) have demonstrated fully autonomous prototypes. These extraordinary applications and demonstrations have motivated research and fueled excitement about the bright future of *Distributed Autonomous*

Mobile Robots (DAMRs).

In this thesis, we tackle the problem of optimizing the energy consumption of DAMR systems from a software perspective. We target a class of DAMR systems where a team of battery-powered and networked robots is navigating in a physical environment and acting in concert to accomplish a common goal. Moreover, we aim to design schedules (aka. plans) for the robots so that they can optimally spend their energy to achieve the goal. To this end, we propose three new problems:

- **Reliability-Energy Tradeoff (RET).** In this problem, we first propose that robots *verify* computational results of one another. By doing this, we can increase the corroboration of outputs of our DAMR systems. However, this new feature requires robots to do additional tasks and thus consume more energy. We then propose approaches to reach a *balance* between *energy consumption* and the *reliability* of results obtained by our DAMR systems.

- **Optimal Recharging (OR).** This problem proposes that our DAMR systems should employ charging station and aims to design schedules for the robots so that they can achieve their goals while consuming as little energy and time (including the time spent for recharging) as possible. Moreover, we model the working area by a connected (possibly incomplete) graph that makes the problem more practical.

- **Optimal Path Planning in the Presence of Disturbances (OPPD).** This path planning problem does not only take into account energy limits but also changes in the physical environment, which may result in overheads (i.e., additional time and energy) that robots incur while doing their tasks. In order to tackle the problem from a software perspective, we propose the use of

Gaussian Process and Polynomial Regression to model disturbances and energy consumption, respectively.

In the rest of this chapter, first, we give some insight into the RET, OR, and OPPD problems in Section 1.1, Section 1.2, and Section 1.3, respectively. Second, we present a review of work related to our research in Section 1.4. Third, we state the contributions of the thesis in Section 1.5. Finally, we describe the outline of the thesis in Section 1.6.

1.1 Reliability-Energy Tradeoff (RET)

Consider a fleet of *unmanned aerial vehicles* (UAVs) that carry out a joint mission, such as search and rescue. The fleet is obviously constrained by energy limits, and a successful completion of the mission depends on rigorous path planning as well as accurate on-the-fly reporting of observations by the individual UAVs. Now, if one of the UAVs is compromised and makes false positive or true negative reports, it may either cause premature completion of the mission or make it unnecessarily long, possibly beyond the current energy limits of the fleet.

Hence the verification feature of DAMR systems (i.e., verifying results obtained from robots) has to be an integral requirement in modern network-connected deployments, especially when they are safety- or mission-critical. Moreover, battery-powered robots usually impose constraints on energy consumption that could potentially affect the level of reliability maintained while executing functional tasks. Hence, DAMR systems should be designed to maximize the level of confidence in task results in the presence of cyberattacks, while reducing energy consumption to extend operational

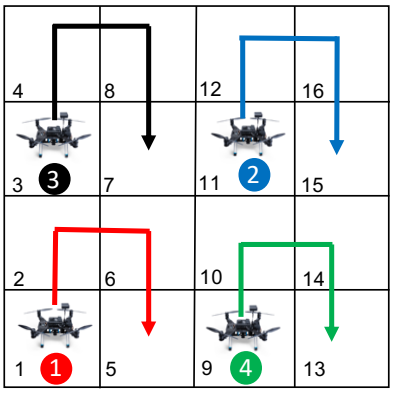
time.

Currently, in most DAMR systems, robots are set to use a certain level of verification at a predefined frequency. Such decisions do not account for the different physical environments where the robots can be deployed, different time-and-space-varying risks, and different threat/vulnerability-levels that call for different measures for reliability. Moreover, most current robotic systems treat verification as a non-functional property and an add-on for the main functional behavior of the robot, and do not account for how verification decisions could affect the control path that the robot takes.

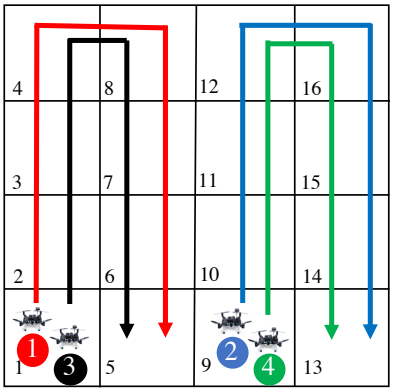
In order to tackle this problem from a software perspective, we introduces a notion of *peer-verification*, where different robots in the system verify the output of each other to gain an acceptable global level of confidence. Obviously, providing more verification results in resource consumption (e.g., energy and communication bandwidth) at higher rates. Thus, one has to reach a balance between resource consumption and the level of reliability guarantees. We propose a general-purpose model that captures the reliability vs. energy tradeoff in DAMR systems and allows system designers to parameterize the tradeoffs in their system. In particular, we consider a global task dependency graph partially executed by mobile robots. Then, we augment the task graph with verification tasks. These tasks can be intermittently and arbitrarily inserted into the computation task graph. When the robots execute more verification tasks, a higher reliability is gained and, in turn, more energy is consumed. Thus, our optimization objective is twofold: maximizing the number of verification tasks executed, and (2) maximizing the number of peers partaking in these tasks. In other words, we need to determine (1) when to schedule verification and (2) which

robots should be involved such that we reach a desired global termination state while operating under per-robot energy constraints.

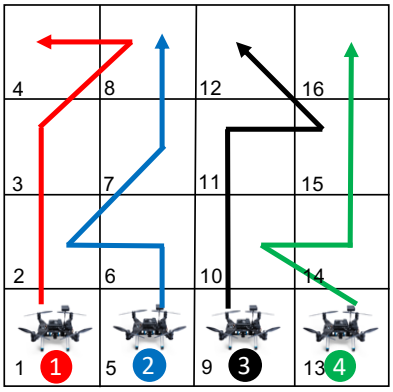
To better explain the idea of optimal peer-verification, consider a multi-UAV search and rescue application. Let us assume that searching each grid cell is a computation task and UAVs in same cells can verify observations of their peers (i.e., doing peer-verifications). Figure 1.1a shows a 4×4 grid area divided to four separate areas and each UAV operates in one of the independent areas with no peer-verification. In this scenario, each UAV has to do four tasks, and the total number of tasks is 16. On the contrary, in Figure 1.1b, extreme verification is enforced by flying two UAVs over each grid cell at all times in two different altitudes. That is, one UAV always verifies the observations of the other one. In this case, the number of tasks done by each UAV is eight, and the total number of tasks is 32. Now, let assume that each all UAVs have equal energy bound, and each is able to do at most six (computation and verification) tasks. This new assumption makes the extreme scenario (i.e., Figure 1.1b) impossible. Figure 1.1c illustrates one of the best plans in which we can have four verification tasks (which is the maximum number of verification tasks that the system can provide, given the UAVs' energy capacity.) In this scenario, UAV 1 (red) and 2 (blue) cover cells 1 – 4 and 5 – 8, respectively, and perform peer-verifications at cells 2 and 8. Likewise, UAV 3 (black) and 4 (green) cover cells 9 – 12 and 13 – 16, respectively, and perform peer-verifications at cells 10 and 15. The total number of tasks done by the four UAVs is 24, and each UAV executes six tasks.



(a) No verification.



(b) Extreme verification.



(c) Optimal 2-peer-verification.

Figure 1.1: Three peer-verification scenarios in a multi-UAV network.

1.2 Optimal Recharging

The impressive advances in theory, design, and deployment of mobile robots and autonomous vehicles in the past decade have brought a wide range of DAMR applications. Delivery companies (DHL International GmbH, 2016; McFarland, 2016; Amazon.com, Inc., 2016; Gatteschi *et al.*, 2015; Wing LLC, 2019), the New York City Police Department (NYPD) (Simon, 2018), military units, agriculture companies, border patrol agencies, etc, now employ DAMR to accomplish various tasks faster and more efficient than humans. However, a shortcoming of DAMR that live on batteries is their severe energy limits. For example, the battery life of state-of-the-art operating UAVs does not go beyond 20 minutes on average. This energy constraint seriously limits the scope of projects and applications that battery-based UAVs can achieve. Furthermore, given the relatively slow progress and lack of breakthroughs in the lithium battery technology, we are in pressing need to design other ways and means to solve the energy restrictions of mobile robots.

One of the plausible approaches proposed to address the energy problem in the context of *smart cities* is to build infrastructure that hosts *charging stations*, where operating mobile robots can recharge their batteries when needed. In the case where a team of robots carry out a joint mission, the underlying research problem can be formulated as a variant of the well-known *vehicle routing problem* (VRP) (Dantzig and Ramser, 1959). VRP is an NP-complete combinatorial optimization problem and is generally concerned with the optimal design of routes by a fleet of vehicles to service a set of customers by minimizing the overall cost, usually the travel distance or energy for the whole set of routes in a weighted directed graph. Although VRP is a widely studied problem, we are only aware of little previous work on VRP, in which

the input graph is augmented with charging stations. Moreover, this line of work lacks the combination of the three key features that need to be addressed to realize sustainable teams of mobile robots in practice: (1) incomplete route graph, (2) multi-objective (i.e., time and energy) optimization, and (3) interaction with a dynamic environment. Specifically, the work in (Conrad and Figliozzi, 2011) allows vehicles to recharge at customer locations instead of designated charging stations, which is an over-simplification in a real infrastructure. Green VRP (GVRP) (Erdoğan and Miller-Hooks, 2012) and electric VRP (EVRP) (Schneider *et al.*, 2014) introduce charging stations, where vehicles can visit multiple times, but their solutions require to know the number of visits to each stations in advance to eliminate subtours. This is also not practical. The variant of VRP with the aforementioned three conditions poses significant new challenges that to our knowledge have not yet been addressed.

With this motivation, we propose an approach to extend the operational time of teams of battery-powered robots by employing charging stations provided by the infrastructure. Our problem setting consists of (1) a team of heterogeneous mobile robots that have different energy limits and can carry out different types of tasks, and (2) a directed connected graph, where the nodes are either a set of goals (or customers) that the team should visit and service a task, or charging stations. We assume that all robots have access to the graph. Since recharging at a charging station takes considerable time, our goal is to design a routing plan that minimizes *both* energy and time use to complete all the tasks in all nodes, including the time spent for recharging.

Although our problem presented in Chapter 4 could alternatively be cast as a variant of the VRP, it has four distinctive features: (1) our route graph is incomplete,

(2) the vehicles are heterogeneous, (3) the number of times that vehicles visit intermediate nodes and charging stations is not required, (4) and the customers of each vehicle can be reassigned due to the changes of physical environment. To the best of our knowledge, we are the first to consider this version of the problem.

1.3 Optimal Path Planning with the Presence of Disturbances

Another critical challenge in designing a DAMR system is to take into account changes in the physical environment in which robots are deployed. For example, wind velocity and direction vary over time, which may not conform with the initial assumption, and possibly cause UAVs to spend more energy and time to follow the pre-computed optimal plans. Although there is a large body of work on the problem (e.g., (Bezzo *et al.*, 2016; Morbidi *et al.*, 2016; Zeng and Zhang, 2017)), most of the existing approaches are from perspectives of system control and avionic, which target to specific types of robots and are difficult to adjust to different types of DAMR. Moreover, there is a lack of work studying the problem from a software perspective.

Path planning problem for DAMR systems has been studied from a software perspective. One of the plausible approaches is to solve extensions of the original VRP which may or may not consider the presence of stationary or mobile obstacles in the working space (e.g., (Krishnan *et al.*, 2017; Pavone *et al.*, 2010; Bullo *et al.*, 2011)). In this line of research, the authors utilize fixed cost functions essentially abstracting away the impact of physical disturbances, which, in reality, can have a profound impact on the local and global energy efficiency of robots.

To address this from a software perspective, we first propose to utilize Gaussian Process and Polynomial Regression to model changes in the physical environment and energy consumption, respectively. Then, we formulate the *optimal path planning in the presence of disturbances (OPPD)* problem, which consists of two fundamental interrelated problems: energy consumption prediction based on disturbances, and path planning.

1.4 Related Work

1.4.1 Reliability of Cyber Physical Systems (CPS)

As our DAMR systems considered in this thesis can be viewed as CPS, we focus on work in increasing the reliability of CPS which has been an increasingly important topic in the last several years. The work in (Cardenas *et al.*, 2009; Sha *et al.*, 2008; Humayed *et al.*, 2017) gives insight into the type of attacks that CPS will be subject to. In order to improve the reliability of CPS, most of the controls discussed in the literature focus on adapting state-of-the-art security techniques to fit within CPS constraints. The work in (Mitchell and Chen, 2014) covers time critical intrusion detection mechanisms usable in a CPS setting. The work in (Fovino *et al.*, 2009) focuses on proposing security-aware alternatives to traditional CPS insecure communication. Out of band authentication was introduced in the scope of medical devices (Rushanan *et al.*, 2014). The work in (Escherich *et al.*, 2009; Wolf and Gendrullis, 2011) introduces hardware-based cryptography solutions specifically designed for smart cars.

One approach relative to the work in this thesis is frequently adding *message*

authentication codes (MACs) to results obtained by systems. The work in (Lesi *et al.*, 2017; Jovanov and Pajic, 2019, 2017) targets the problem of determining when to inject cryptographic checks without interfering with control tasks; they propose an approach that tries to maximize reliability checks while maintaining a predefined level of control quality. Similarly, the work in (Xia *et al.*, 2008) proposes a feedback scheduling technique for maintaining network QoS in wireless sensor networks.

Comparison to our work. The primary difference between our RET problem, presented in Chapter 3, from work mentioned above is that we tackle the problem from a scheduling perspective. That is, we consider the verification feature of CPS as optional verification tasks (having robots verify results of one another), incorporate them within the existing task graph, and manage to assign all compulsory tasks and as many optional verification tasks as possible to robots under energy constraints. This approach is more flexible than the existing ones because the reliability level of the system (i.e., the number of executed verification tasks and the number of peers partaking these tasks) is determined at the deployment time and adjusted regarding the physical environment where robots are deployed.

1.4.2 Task Scheduling

One of the major challenges in the RET problem proposed in this thesis is concerned with the optimal schedule of tasks for the team of robots such that they can finish all computation tasks, which are compulsory, and execute as many as possible of the optional verification tasks. In other words, if the computation and verification tasks are simply viewed as compulsory and optional ones, respectively, then the problem can be reduced to the task scheduling problem, in parallel and distributed computing

systems. That is, given such a system, a set of dependent or independent tasks and a set of constraints, the ultimate goal of the problem is to find the assignment of tasks to processors that will let the system complete the tasks and satisfy all the constraints in the shortest time. The problem is shown to be NP-complete in (El-Rewini *et al.*, 1995). The work in (Wu *et al.*, 2004; Wu and Gajski, 1990; Ahmad and Kwok, 1998) considers homogeneous parallel and distributed systems (i.e., the processors are identical) and proposes various heuristic algorithms to solve the problem. It becomes more challenge if the systems are heterogeneous (i.e., they have various processors.) Therefore, the authors in (Wang *et al.*, 1997; Kwok and Ahmad, 1996; Sih and Lee, 1993; El-Rewini and Lewis, 1990; Kruatrachue and Lewis, 1988; Hwang *et al.*, 1989) propose two-phase heuristic algorithms to solve the problem for such systems. First, tasks are ordered and selected based on their priorities. Then the ordered tasks are allotted to the most suitable processor to minimize the execution time.

Comparison to our work. A shortcoming of the aforementioned work is not to take into account the cost of transition between the given tasks. For example, there are two tasks t_1 and t_2, and t_2 needs the results from t_1. This means, t_2 can only be started when t_1 is done, and it takes time to transfer data from the processor doing t_1 to the one executing t_2 if the tasks are run on two different processors. This cost will be negligible if the system under scrutiny is not very large (e.g., multiple processors within a single server, multiple servers within a local network.) Otherwise, it will be significant, especially when the processors are geographically distant and connected by a wide area network. To rigorously address this, in our RET problem, we consider both execution and transition costs.

Additionally, the major goal of our RET problem is to increase the sanity of the

system under energy constraints. That is, our optimization is not to minimize the global cost to finish all the given tasks but to maximize the number of verification tasks as well as peer-verification while remaining in the system energy bounds.

1.4.3 Mobile Robots Planning

There is an extensive line of work on the paths and trajectories planning problem for multiple mobile robots in both centralized and decentralized manners. Regarding to the former approach, the work in (Erdmann and Lozano-Perez, 1987; Saha *et al.*, 2014, 2016; Van Den Berg and Overmars, 2005) proposes a solution requiring a central server to compute the collision-free trajectories for a fleet of homogeneous robots with the presence of stationary obstacles in the workspace. The work in (Kim *et al.*, 2013; Kim and Morrison, 2014; Song *et al.*, 2014a,b) introduces a new variant of the problem in which mobile robots can visits charging station to recharges their batteries in order to extend their operational time and pursuit long-term task. However, the authors require that each task must be split-able into smaller jobs, which is not always achievable.

There is also a line of research addressing the problem in a decentralized manner. The authors in (Turpin *et al.*, 2014) propose two different algorithms to solve the problem in both centralized and distributed manners. However, the decentralized version requires that the number of task must be equal to the number of robots and there does not exist any obstacle in the working environment. Although the work in (Cáp *et al.*, 2013; Guo and Parker, 2002; Velagapudi *et al.*, 2010) proposes only decentralized solutions, the algorithms allow the presence of stationary obstacle in the environment under scrutiny. The work in (Desai *et al.*, 2017; Zimmerman, 2013;

Lin and Mitra, 2015) provides provably correct distributed path planning for mobile robots.

Comparison to our work. In our work, we propose a novel formulation of a task assignment and path planning problem for a team of heterogeneous robots that aims at optimizing energy consumption as well as the time needed to complete the tasks under energy limitations. Moreover, in Chapter 4 and 5, we assume that our working environment can be modeled by a *connected but not complete* graph whose nodes are customers, where the robots have to arrive to do some tasks, or charging stations, where they can recharge their batteries, and arcs denote the possibility of traveling recharging. With this assumption, our proposed system is more practical and easily applied to real-world scenarios.

1.4.4 Path Planning

One of the major challenges in our research is concerned with the optimal design of paths used by the fleet of the robots to perform their tasks located different locations. This problem can be reduced to the *Traveling Salesman Problem* (TSP) that aims to find the cheapest route for a salesman so that he can visit all his customers, and return home (aka. the depot.) The TSP is one of the most studied combinatorial optimization problems and has many variants. In the scope of this thesis, we focus on two specific variants:

- *Multiple TSP (mTSP).* In this variant, there are multiple identical salesmen, and the goal is to find the routes for them to minimize the global traveling cost. Applications of the problem in DAMR can be found in (Hartuv *et al.*,

2018; Park and Morrison, 2014) in which the authors design a set of routes for a team of homogeneous UAVs to monitor different parts of an area. Moreover, the route for each UAV must not exceed the energy bound of the UAV.

- *TSP for road network (R-TSP).* Although this variant considers only a single salesman, the transitional capability between his customers is modeled by a sparse (connected but incomplete) graph. That is, there may not be a direct connection between a pair of customers, and the salesman inevitably visits one or more intermediate places while going from one to the other. This problem was first introduced in (Fleischmann, 1985) and (Cornuéjols *et al.*, 1985). The work in (Letchford *et al.*, 2013) and (Interian and Ribeiro, 2017) proposes a polynominal size mixed integer linear programming formulation of the problem and a heuristic algorithm, respectively.

Another approach to solving the path planning problem is to reduce it to the VRP (Dantzig and Ramser, 1959). The problem takes as input a set of vehicles and a set of customers who need to be serviced by the vehicles, and the goal is to design routes for the vehicles such that all the customers are serviced, and the global cost is minimized. The VRP is also one of the extensively studied combinatorial optimization problems. A survey of the problem and its variants can be found in (Kumar and Panneerselvam, 2012). However, the literature on the problem with charging stations is still relatively thin. The work in (Conrad and Figliozzi, 2011), to the best of our knowledge, was the first to consider a variant of VRP in which vehicles are allowed to optionally recharge at customer locations. Due to the fact that (1) the problem is modeled by a complete graph, (2) each customer is visited once, and (3) vehicles can be recharged at any customers, the solution of the problem does not contain any cycle

(so it is relatively easier to solve than ours.) Erdogan and Miller-Hooks (Erdoğan and Miller-Hooks, 2012) propose the green VRP (G-VRP) in which vehicles run on fuel and can be refueled at fuel stations (i.e., charging stations). Although the problem, similar to other variants of VRP, is modeled by a complete graph, each fuel station may be visited more than once or not at all. This means that routes assigned to the vehicles may have cycles, but the authors require to know the number of times n_f that each fuel station v_f is visited in advance. Unfortunately, determining n_f's is not trivial because they should be set as small as possible to reduce the network size but large enough not to restrict multiple beneficial visits. The work in (Schneider *et al.*, 2014) proposes Electric VRP (EVRP) which is an extension of G-VRP focusing on electric vehicles (EVs) and with time windows (i.e., customers must be serviced within a given time interval.) The authors also require to know the number of times that a charging station is visited to eliminate subtours.

Comparison to our work. Our work also tackles the path planning problem, but our problem differs from the aforementioned problems in the following two aspects:

- In Chapter 4, our OR problem is a generalization of the aforementioned problems. That is, our OR problem aims to design a set of optimal routes for multiple heterogeneous vehicles in a sparse (i.e., connected but not complete) graph. Additionally, our vehicles have limited traveling resource (i.e., energy) which can be replenished at some specific nodes (i.e., charging station.) Moreover, our proposed integer programming transformation and algorithms do not require n_f as input parameters (i.e., the number of times that each charging station v_f is visited).

- In Chapter 5, our OPPD problem takes into account changes in the physical environment which cause the traveling and service costs (i.e., energy and time) vary over the time.

1.5 Contributions

Our research goal is to show that *exclusively software-based approaches can provide an effective energy-optimization for DAMR systems.* By effective, we mean that our systems can be actually implemented and deployed in real-world scenarios. To this end, we study a class of DAMR systems where the team of battery-powered and networked robots navigating in a physical environment and acting in concert to accomplish a common goal. Our main contributions are the following:

- We introduce a notion of *peer-verification*, where different robots in the system verify the output of each other to gain acceptable global confidence. Obviously, providing more reliability results in higher energy consumption, and one has to reach a balance between energy consumption and the level of reliability guarantees. Thus, we propose a general-purpose model that captures the interactions of the reliability vs. energy tradeoff in the DAMR systems, and allows system designers to parameterize the tradeoffs in their systems. To the best of our knowledge, this is the first work doing this.

- We propose an approach to extend the operational time of the DAMR by deploying charging stations where the robots can come to recharge their batteries. Moreover, we constrain the robots to follow specific routes to move from one point to another in the a priori known working environment. Technically, the

working environment is modeled as a directed, connected, and finite graph whose nodes are charging stations or customers (where the robots have to come to do some tasks), and arcs denote the possibility of traveling. We propose a novel problem combining task assignment and path planning aiming at optimizing energy consumption as well as the time needed to complete the tasks, including the time spent for recharging.

- We propose an approach to plan missions for a team of robots working in the presence of disturbances. The problem is a combination of two fundamental interrelated problems: (1) predicting energy consumption, and (2) and path planning for a team of robots with the presence of disturbances in their working environment. Remind that our proposed algorithms also deal with the energy limitation of each robot. To address the problem from a software perspective, we propose the use of Gaussian Process and Polynomial Regression to model disturbance dynamics and energy consumption, respectively. Moreover, we also propose a decentralized algorithm based on negotiation to solve the problem online.

- We fully implemented our algorithms and report results of not only simulation but also *experiments on a real network of UAVs*. Moreover, we would like to emphasize that in our experiments, when a UAV flew below another one, it consumed 12% additional energy to resist the spiral airflow caused by the higher UAV. These overheads are obscure and are ignored by our offline algorithms but not our online algorithms.

- In order to conduct the experiments, we developed a *Robot Operation System (ROS)* package (i.e., ROS application) which was executed on each UAV, allowed UAVs to extract their location information from our motion capture system, and let them stably fly in our lab space.

1.6 Thesis organization

This thesis consists of six chapters and tackles three different aspects of optimizing energy for the DAMR systems:

- Chapter 1 informally describes our problems and contributions.
- In Chapter 2, the preliminary concepts related to this thesis is introduced.
- Chapter 3 formally defines the Reliability-Energy Tradeoff (RET) problem. The problem is formulated as an *integer linear program (ILP)* which can be utilized to obtain optimal solution for relatively small size problems. Thus, we also proposes other heuristic techniques to address the practical problems in both offline and online manners. The experimental results of our proposed algorithm are presented at the end of the chapter.
- In Chapter 4, we formally define the Optimal Recharging (OR) problem that aims to utilize charging stations to extend operational time of robots and formulated it as a multi-objective integer programming (MIP). Additional, we propose heuristic algorithms to solve the problem in both offline and online manners. The experimental results of our proposed algorithm are presented at the end of the chapter.

- Chapter 5 tackles the path planning with the presence of disturbances which involves both the problems of predicting energy consumption and path planning. and propose techniques to address it. In this chapter, we propose the use of Gaussian Process, and Polynomial Regression to model disturbances dynamics and energy consumption, respectively. With these tools, we formulate the problem as a multi-graph and an ILP. We also propose heuristic algorithms to solve the problem in both offline and online manners. The experimental results of our proposed algorithm are presented at the end of the chapter.

- Finally, in Chapter 6, we conclude and present possible future work.

Chapter 2

Preliminary Concepts

In this chapter, we present the preliminary concepts including our notion of *verificable task graphs*.

2.1 Task Graph

To represent a set of tasks, which will be assigned and executed by the robots, and their *partial order*, we utilize a *directed acyclic graph* (DAG). Formally,

Definition 1. *A* task graph *is a DAG* $G = (V, E)$*, where each vertex* $t \in V$ *is a* task*, and each arc* (t, t') *in* E *specifies an* order*, where task* t' *depends on* t *(that is, task* t *must be finished before task* t' *can be executed.)* □

In a task graph $G = (V, E)$, we denote the set of *predecessors* (respectively, *successors*) of a task t by P_t (respectively, S_t.) Formally,

$$P_t = \{t' \mid (t', t) \in E\} \qquad S_t = \{t' \mid (t, t') \in E\} \tag{2.1.1}$$

An *entry point* is a task with no incoming edges, and a *termination point* is a task with no outgoing edges. For example, Figure 2.1 demonstrates a task graph of seven tasks, where t_1 and t_2 are entry points, and t_6 and t_7 are termination points.

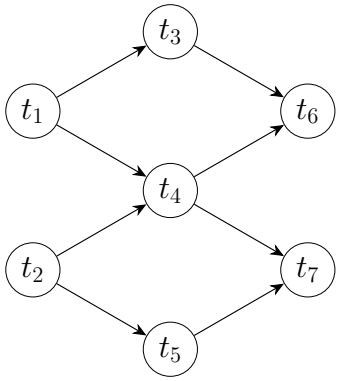


Figure 2.1: Task graph.

2.2 Verificable Task Graph

We expand Definition 1 by augmenting task graphs with means for verification. We define two equal-size sets of correlated tasks:

- *Computation tasks* (denoted $V_T = \{t_1, \ldots, t_k\}$) that represent the actual work to be done. These are the tasks in Definition 1.
- *Verification tasks* (denoted $V_A = \{\alpha_1, \ldots, \alpha_k\}$) associated with computational tasks. An verification task *optionally* follows the actual work and verifies the output of a computational task.

Definition 2. *Let $G = (V, E)$ be a task graph as defined in Definition 1. The corresponding* verificable task graph *is a DAG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where*

$$\mathcal{V} = V_T \cup V_A$$

is the set of all tasks. The directed edges in the graph are as follows:

- *For each edge $(v_i, v_j) \in E$, we include an edge (t_i, t_j) in $\mathcal{E}$ to retain the dependency.*
- *There is an edge from a computation task to its corresponding verification task. That is,*

$$\forall t_i \in V_T : (t_i, \alpha_i) \in \mathcal{E} \tag{2.2.1}$$

□

Figure 2.2 illustrates the verificable version of the task graph in Figure 2.1. As can be seen in the figure, for every task t_i, there is a corresponding verification task α_i. The dashed arrows represent the edges from computation tasks to their corresponding verification tasks.

2.2.1 Route graph

In real-world environments, an autonomous mobile robot (such as a self-driving car) is not always possible to move directly from one place to another but has to follow specific paths (or streets) and inevitably visit one or more intermediate places (cities)

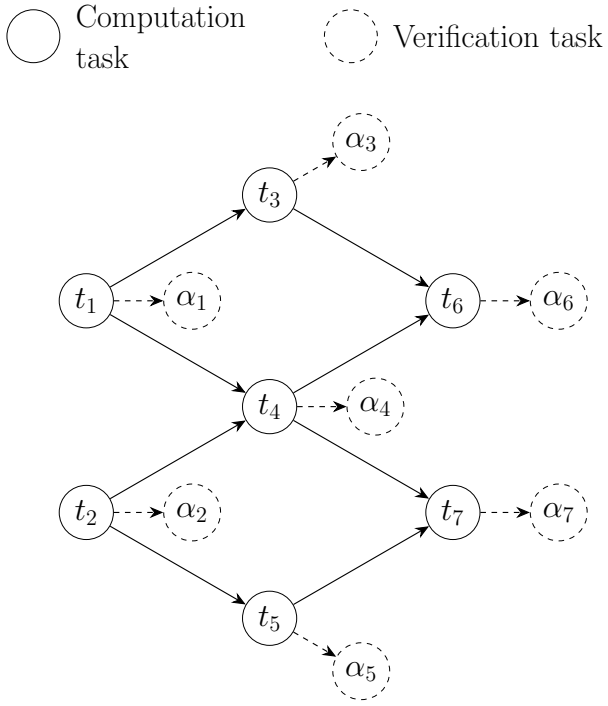


Figure 2.2: The verificable task graph of the task graph in Figure 2.1.

before reaching its destination. In an effort to mimic the original environment, we employ a directed, connected, and finite graph named *route graph*. Formally,

Definition 3. *A* route graph *is a directed, connected, and finite graph* $G = (V, E)$*, where:*

- $V = \{v_0\} \cup V_C \cup V_B$ *is the set of all vertices, where* v_0 *is the* depot*,* V_C *is the set of customers where a robot have to come to do a task, and* V_B *is the set of (battery) charging stations where a robot can come to recharge its battery, such that* $v_0 \notin V_C \cup V_B$*, and* $V_C \cap V_B = \emptyset$ *(i.e., each vertex can play only a single role, being either the depot, a customer, or a charging station.)*
- E *is a set of arcs, where an arc* $(v, v') \in E$ *denotes the possibility of traveling from node* v *to* v'*.* □

Figure 2.3 illustrates an example of a route graph. For simplicity, we utilize an undirected edge (v, v') to replace two arcs (v, v') and (v', v). As can be seen in the figure, (1) the graph is connected but incomplete, (2) the depot v_0 locates at the bottom left of the graph, (3) and there are six customers (i.e., c_1, c_2, c_3, c_4, c_5, and c_6) and two charging stations (i.e., b_1 and b_2).

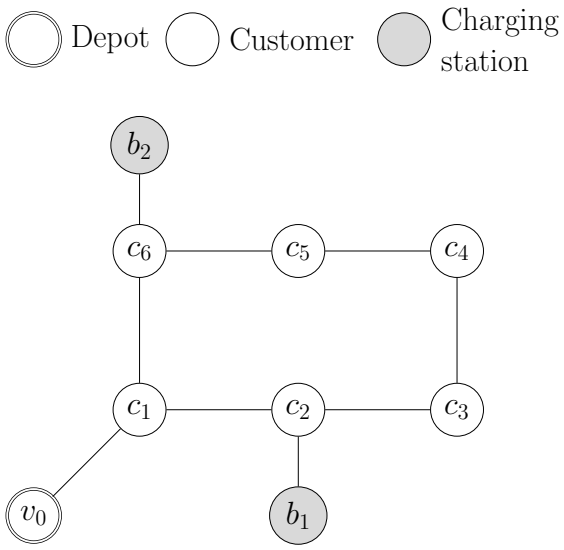


Figure 2.3: route graph

2.3 Execution unit

We view each robot as an execution unit that is able to execute the tasks and is constrained by its energy capacity. For example, UAVs are able to perform certain tasks within their energy limits and they consume some minimal energy when they are idle.

Definition 4. *An* execution unit $u = \langle \mathcal{A}_u, \mathcal{B}_u, \mathcal{I}_u \rangle$ *is a tuple of three elements:*

- *The* affinity set $\mathcal{A}_u$ *is a set of tasks that* u *is allowed to execute;*

- Energy bound $\mathcal{B}_u$ *is a non-negative real number indicating the upper bound on the energy of* u*;*
- Idle energy consumption rate $\mathcal{I}_u$ *is a non-negative real number indicating the amount of energy consumed every second when the execution unit is idle, i.e. not executing any task.*

□

We denote the set of all execution units by $\mathbb{U}$. An execution unit consumes time and energy when executing a task. There is also cost associated with transitioning from one task to another. For instance, in the case of UAVs, there may be geographical separation between tasks and so the order of tasks determines the flight distance and consequently the required amount of time and energy to execute the tasks.

Chapter 3

Reliability-Energy Tradeoff

In this chapter, we deal with the RET problem which aims to control energy consumption in order to improve reliability of our DAMR systems (a brief introduction of the problem was presented in Section 1.1). To this end, we first formalize the problem as a task scheduling problem. Unlike existing task scheduling problems, our task graph, called *verificable task graph* (see Section 2.2), contains not only compulsory computation tasks but also optional verification task. Then, in order to find optimal peer-verification strategies, in which all compulsory tasks and as many as possible of optional verification tasks are executed, we investigate four offline and online techniques:

- First, we propose an *offline* optimization method that produces a schedule of computation tasks intermittent with verification tasks. This method is based on transforming our problem into *integer linear programming* (ILP). This technique produces the optimal result, though is limited to small problem sizes.

- Next, we introduce two scalable *offline* algorithms that find sub-optimal solutions. The first algorithm is a non-trivial greedy technique, where execution units run and authenticate the tasks with the least energy requirements. To ensure that we complete all mandatory tasks in the graph, the algorithm checks to see if there is sufficient energy to execute all remaining computation tasks (which are compulsory) without verification. At the cutoff point, the algorithm will abandon verification and execute all remaining computation tasks only. We also propose a genetic algorithm that generates a random population (i.e., sequence of task executions). The population is then evolved by mating, mutating, selecting its individuals. Upon reaching the terminating condition, the best individual in the population is selected as the final solution to the problem.

- Since physical environments inherently bring uncertainties in our DAMR systems, offline solutions are often not able to react properly to new energy and/or reliability conditions on the ground. Thus, we propose an *online* algorithm that schedules tasks dynamically according to current energy limits of execution units by using backtracking to the last feasible step and resolving the optimization problem.

We have fully implemented our algorithms and report results of simulation as well as experiments on a real network of UAVs. Our simulations show that our genetic algorithm consistently outperforms the greedy algorithm in terms of executing more verification tasks and better utilizing the remaining energy of execution units, while the ILP-based technique clearly finds the optimal solutions. We also develop a proof of concept by deploying our algorithms in a real network of UAVs that carry out a joint search mission. We also present the results of experiments for our online

algorithm in a scenario, where UAVs consume more energy than predicted, and how the algorithm successfully navigates them to complete their mission within their real limits, while maintaining the best reliability.

The rest of this chapter is organized as follows. Section 3.1 is to introduce the formal statement of the RET problem. Next, in Section 3.2, we transform our problem into ILP. Then, in Sections 3.3, 3.4, and 3.5 we present our greedy, genetic, and online algorithms, respectively. Finally, we report our simulation and experimental results in Section 3.6

3.1 Problem Statement

In this section, we formally state our RET problem. To this end, we start by considering costs of an execution unit, and then identify constraints and objectives of the problem.

3.1.1 Cost Functions

To determine the costs of an execution unit $u \in \mathbb{U}$ executing tasks in an verificable task graph $\mathcal{G}$, we define the following cost functions:

- *Execution cost* $\xi : \mathbb{U} \times \mathcal{V} \to \mathbb{R}^2_{\geq 0}$ maps an execution unit u and a (computation or verification) task v to a pair of real numbers (τ, ε), where τ and ε are the amount of time and energy consumed by the unit executing v.

- *Transition cost* $\mathcal{T} : \mathbb{U} \times \mathcal{V}^2 \to \mathbb{R}^2_{\geq 0}$ is a function that maps an execution unit u and a pair of tasks v, v' to a pair of real numbers (τ, ε), where τ and ε are the amount of time and energy consumed by the unit transitioning from v to v'.

3.1.2 Schedule

A schedule $\mathcal{S}$ is a function that determines which execution units run a task and when they start executing it. It is defined as follows:

$$\mathcal{S} : \mathcal{V} \rightarrow 2^{\mathbb{U}} \times \mathbb{R}_{\geq 0}$$

Thus, $\mathcal{S}$ maps a task in the set of vertices $\mathcal{V}$ in verificable graph $\mathcal{G}$ to a set of execution units $\mathbb{U}$ and a start time $s \in \mathbb{R}_{\geq 0}$.

Note that the schedule allows assigning a task to multiple execution units and only one start time. The purpose of this is to support verification tasks. An verification task is performed by two or more units concurrently. This allows us to model communication as part of the peer-verification process such as key exchange. That is, peer-verification should take place at the same time among peers. It also allows us to model corroboration of a task output such as partial replication of the computation or verification of digests. A computation task on the other hand can only be executed by one execution unit. We enforce these constraints in the next subsection.

An *assignment* $p_u^{\mathcal{S}}$ is the ordered sequence of tasks executed by u as determined in schedule $\mathcal{S}$. It is defined as $p_u^{\mathcal{S}} = (\upsilon_0, \upsilon_1, \cdots)$ such that

$$\begin{aligned} \forall \upsilon \in p_u^{\mathcal{S}} &: u \in \mathcal{S}(\upsilon_i).U \\ \forall \upsilon_i, \upsilon_{i+1} \in p_u^{\mathcal{S}} &: \mathcal{S}(\upsilon_i).s < \mathcal{S}(\upsilon_{i+1}).s \end{aligned} \tag{3.1.1}$$

where $\upsilon_i \in \mathcal{V}$, and $\mathcal{S}(\upsilon).U$ and $\mathcal{S}(\upsilon).s$ denote the set of units and start time mapped to υ.

Given an verificable task graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a set of execution units $\mathbb{U}$. The goal of the RET problem is to design a schedule $\mathcal{S}$ whose constraints and objectives are as follows:

3.1.3 Problem Constraints and Optimization Objectives

Dependency. A schedule obeys the dependency encoded in $\mathcal{G}$, such that the following holds:

$$\forall (v, v') \in \mathcal{E} : \mathcal{S}(v').s \geq \mathcal{S}(v).s + \max\{\xi(u, v).\tau \mid u \in \mathcal{S}(v).U\} \tag{3.1.2}$$

That is, for every pair of tasks v and v', where v' depends on v, the start time of v' is greater or equal to the termination time of v. The termination time of v is calculated as the start time of v plus the execution time of v spent by its assigned execution unit as determined by the schedule.

Transition time. A schedule assigning a sequence of tasks to an execution unit must account for the transition time:

$$\begin{aligned} &\forall u \in \mathbb{U} : \forall (\upsilon_i, \upsilon_{i+1}) \in p_u^{\mathcal{S}} : \\ &\qquad \mathcal{S}(\upsilon_{i+1}).s \geq \mathcal{S}(\upsilon_i).s + \xi(u, \upsilon_i).\tau + \mathcal{T}(u, \upsilon_i, \upsilon_{i+1}).\tau \end{aligned} \tag{3.1.3}$$

Affinity. Every task is run by an execution unit allowed to run the task as determined by its affinity set $\mathcal{A}_u$:

$$\forall v \in \mathcal{V} : v \in \mathcal{A}_{\mathcal{S}(v).U} \tag{3.1.4}$$

where $\mathcal{A}_{\mathcal{S}(v).u}$ is the affinity set of the execution unit assigned to run v as determined

by the schedule $\mathcal{S}$.

Completion. All computation tasks must be executed exactly once.

$$\forall t \in T : |\mathcal{S}(t).U| = 1 \tag{3.1.5}$$

Energy bound. Each execution unit should obey its energy bound. That is, for every execution unit, the total amount of energy consumed while executing an assignment is less than or equal to its bound. First, note that an execution unit can also be idle between tasks. For an assignment $p_u^{\mathcal{S}}$, let $\iota(p_u^{\mathcal{S}}, i)$ denote the idle time spent before executing the ith task in the assignment:

$$\iota(p_u^{\mathcal{S}}, i) = \mathcal{S}(v_i).s - \mathcal{S}(v_{i-1}).s - \xi(u, v_{i-1}).\tau - \mathcal{T}(u, v_{i-1}, v_i).\tau \tag{3.1.6}$$

Thus, the following constraint enforces that an execution unit obeys its energy bound:

$$\forall u : \sum_{i=0}^{|p_u^{\mathcal{S}}|-1} \xi(u, v_i).\varepsilon + \mathcal{T}(u, v_i, v_{i+1}).\varepsilon + \mathcal{I}_u(\iota(p_u^{\mathcal{S}}, i+1)) \leq \mathcal{B}_u \tag{3.1.7}$$

Peer verification. As mentioned earlier, an verification task is performed by two or more units concurrently. First, we enforce that two or more units participate in every verification task as follows:

$$\forall \alpha \in A : |\mathcal{S}(\alpha).U| = 0 \vee |\mathcal{S}(\alpha).U| \geq 2 \tag{3.1.8}$$

That is, if an verification task is ever executed, it must be executed by at least two peers. Second, we enforce that one of the execution units participating in verification is the unit that performed the respective computation task. This allows us to model

verification that depends on the task's output, which is expected when corroborating the output or reaching consensus:

$$\forall \alpha_i \in \left\{\alpha \mid |\mathcal{S}(\alpha).U| \neq 0\right\} : \mathcal{S}(t_i).U \subset \mathcal{S}(\alpha_i).U \tag{3.1.9}$$

where t_i is the respective computation task of α_i. That is, for all verification tasks that were executed per the schedule, the set of execution units that executed their respective computation tasks (which is only one unit as per Constraint (3.1.5)) is a subset of the verification peers.

3.1.4 Optimization Objectives

More verification tasks implies higher confidence in task results. Our optimization objective is twofold:

- *Maximize verifications.* Our first objective is to maximize the number of verification tasks executed by any execution unit. That is:

$$\max \left| \left\{\alpha \mid |\mathcal{S}(\alpha).U| \neq 0\right\} \right| \tag{3.1.10}$$

  In this formula, every verification task is counted once regardless of how many peers participated in verification.

- *Maximize peers.* A higher number of peers participating in verification also increases confidence and resilience to attacks. Thus, our second objective is to

maximize the average number of peers per verification task:

$$\max \frac{\sum_{\alpha \in A} |\mathcal{S}(\alpha).U|}{\left|\left\{\alpha \,\middle|\, |\mathcal{S}(\alpha).U| \neq 0\right\}\right|} \tag{3.1.11}$$

3.2 Integer Linear Program Transformation

As the first step to solve the optimization problem presented in Section 3.1, we transform it into a ILP.

3.2.1 Decision Variables

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be an verificable task graph, $\mathcal{T}$ be the cost function, and $\mathbb{U}$ be the set of all execution units. The following are decision variables in our ILP instance:

- $x_i^u \in \{0, 1\}$ indicates whether an execution unit $u \in \mathbb{U}$ executes task i, where $i \in \mathcal{V}$.
- $y_i \in \{0, 1\}$ indicates whether multiple execution units execute verification task i, where $i \in V_A$.
- $x_{ij}^u \in \{0, 1\}$ indicates whether an execution unit $u \in \mathbb{U}$ transitions from task i to j, where $i, j \in \mathcal{V}$.
- $s_i \in \mathbb{R}_{\geq 0}$ is the start time of task i, for $i \in \mathcal{V}$. Note that since there is only one start time for each task, if an verification task is executed by multiple units, it will be executed concurrently.
- $f^u \in \mathbb{R}_{\geq 0}$ indicates the time when execution unit $u \in \mathbb{U}$ finishes the last task in its assignment, for $i \in \mathcal{V}$.

- $l^u \in \mathbb{R}_{\geq 0}$ indicates the total amount of idle time spent by execution unit $u \in \mathbb{U}$ when processing its assignment.

3.2.2 Constraints

This subsection details the constraints of the ILP instance based on the ones identified in Section 3.1.3.

Affinity. An execution unit can only execute tasks in its affinity set:

$$\sum_{i \notin \mathcal{A}_u} x_i^u = 0 \qquad \text{for all } u \in \mathbb{U} \tag{3.2.1}$$

Completion. Every computation task in the graph must be executed once:

$$\sum_{u \in \mathbb{U}} x_i^u = 1 \qquad \text{for all } i \in V_T \tag{3.2.2}$$

Dependency. First, we assume a dummy task d_s which is utilized to connect all entry points in $\mathcal{G}$. Formally,

$$\forall v \in \{v \mid |P_v| = 0\} : (d_s, v) \in \mathcal{E}' \wedge \sum_{u \in \mathbb{U}} \mathcal{T}(u, d_s, v) = (0, 0)$$

where $\mathcal{E}' \supset \mathcal{E}$. Moreover, every execution unit has to initially execute d_s with no time or energy cost. That is,

$$\begin{aligned} s_{d_s} &= 0 \\ x_{d_s}^u &= 1 && \text{for all } u \in \mathbb{U} \\ \xi(u, d_s).\tau = \xi(u, d_s).\varepsilon &= 0 && \text{for all } u \in \mathbb{U} \end{aligned}$$

Now we constrain a task's start time to enforce dependency:

$$s_j \geq s_i + \sum_u x_i^u \times \xi(u,i).\tau \quad \text{for all } (i,j) \in \mathcal{E}'$$

Consecutiveness. We constrain that an assignment is a consecutive sequence which starts from d_s, goes through adjacent tasks and does not return to d_s. That is,

$$\sum_{j \in \mathcal{V}} x_{d_s j}^u = 1 \qquad \text{for all } u \in \mathbb{U}$$

$$0 \leq \sum_{i \in \mathcal{V} \cup \{d_s\}} x_{ij}^u - \sum_{k \in \mathcal{V}} x_{jk}^u \leq 1 \qquad \text{for all } u \in \mathbb{U}, j \in \mathcal{V}$$

$$\sum_{i \in \mathcal{V}} x_{i d_s}^u = 0 \qquad \text{for all } u \in \mathbb{U}$$

Transition time. Next, we ensure that the start time of tasks accounts for the transition time from the previous task:

$$s_j \geq M(x_{ij}^u - 1) + s_i + x_{ij}^u \times ((\xi(u,i).\tau + \mathcal{T}(u,i,j).\tau) \qquad \text{for all } i,j \in \mathcal{V}, u \in \mathbb{U} \tag{3.2.3}$$

where M is a large constant (larger than maximum possible start time) to cancel the constraint if unit u does not execute i then j.

Energy bound. Every execution unit honors its energy bound. To constrain energy consumption, we first add constraints to bound the finish time of execution units:

$$f^u \geq M(x_i^u - 1) + s_i + \xi(u,i).\tau \qquad \text{for all } u \in \mathbb{U},\ i \in \mathcal{V} \tag{3.2.4}$$

Thus, f^u is greater than or equal to the finish time of the last task that execution unit u runs. Next, l^u is the idle time spent by unit u:

$$l^u = f^u - \sum_{i \in \mathcal{V}} x_i^u \times \xi(u,i).\tau - \sum_{(i,j) \in \mathcal{V}^2} x_{ij}^u \times \mathcal{T}(u,i,j).\tau$$

Finally, we constrain the energy consumption of execution unit u as follows:

$$\sum_{i \in \mathcal{V}} x_i^u \times \xi(u,i).\varepsilon + \sum_{(i,j) \in \mathcal{V}^2} x_{ij}^u \times \mathcal{T}(u,i,j).\varepsilon + l_u \times \mathcal{I}_u \leq \mathcal{B}_u$$

$$\text{for all } u \in \mathbb{U}$$

Peer Verification. An verification task must be executed by two or more execution units.

$$\begin{aligned} &\sum_{u \in \mathbb{U}} x_i^u \leq 0 + y_i \, M \\ &\sum_{u \in \mathbb{U}} x_i^u \geq 2 + (y_i - 1) \, M \end{aligned} \qquad \text{for all } i \in A \tag{3.2.5}$$

where $M > |\mathbb{U}|$.

Verification after computation. An verification task must follow its respective computation task in order to verify the output of the computation. Essentially, we want to ensure that the arc from a computation task to its respective verification task in the verificable graph is visited by a unit participating in the verification.

$$\sum_{u \in \mathbb{U}} x_{ij}^u \geq y_j \qquad \text{for all } (i,j) \in \mathcal{E} \wedge j \in V_A \tag{3.2.6}$$

3.2.3 Objective Functions

As mentioned in Subsection 3.1.4, our problem requires optimizing multiple objectives. First of all, our main objective is to maximize the number of executed verification tasks, regardless of the number of units participating in verification. Hence, our first objective is as follows:

$$\max \sum_{i \in V_A} y_i \tag{3.2.7}$$

Upon solving the ILP given Objective (3.2.7), we add a constraint as follows:

$$\sum_{i \in V_A} y_i = \mathbb{M} \tag{3.2.8}$$

where $\mathbb{M}$ is the solution of the ILP with Objective (3.2.7). Based on the new constraint, we set the objective of the new ILP to the following:

$$\max \sum_{u \in \mathbb{U}} \sum_{i \in V_A} x_i^u \tag{3.2.9}$$

Thus, the new ILP maximizes the number of participants in verification tasks given the number of executed verification tasks is $\mathbb{M}$.

3.3 Greedy Algorithm

In this section, we introduce our greedy algorithm (Algorithm 1). The greedy choice in the algorithm is running and authenticating the tasks with the least energy requirements. To ensure that we complete all tasks in the graph, the algorithm checks

to see if there is available energy to execute all remaining computation tasks without verification. At the cutoff point, the greedy algorithm will abandon verification and execute all remaining computation tasks only.

Our algorithm implements the Nearest Neighbor Heuristic (NNH) (i.e., computation tasks are sequentially chosen according to their energy requirements) to check whether there are feasible assignments to execution units that result in the completion of all remaining tasks. It takes as input an verificable task graph $\mathcal{G}$, a set of execution units $\mathbb{U}$, and the partially built assignments to the units $\tilde{\Lambda}$. The algorithm works as follows:

1. Determine the set of available computation tasks $\bar{T}$ (i.e., the set of computation tasks that have not been executed, and whose all predecessors have been completed) (Line 1).

2. Utilize a loop to sequentially extend the assignment of each unit (Lines 2 -14). At each iteration,

   (a) choose the cheapest task to execute (Line 8)

   (b) if the unit has enough energy to do the task, append the task to the end of the unit's assignment (Line 10) which potentially unlock other tasks that were waiting for their precedents to complete. These tasks can now be added to $\bar{T}$ (Line12)

   (c) otherwise, switch to the next execution unit (Line 14).

3. If at the end of the loop there are still unexecuted tasks, this indicates a failure to explore the graph with the available energy. Otherwise, succeed.

Algorithm 1: GreedilyExplore

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $\mathbb{U}$, $\tilde{\Lambda}$
Output: **True** or **False**

1 $\bar{T} \leftarrow$ set of available computation tasks derived from $\tilde{\Lambda}$
2 **foreach** $u \in \mathbb{U}$ **do**
3 *finished* $\leftarrow$ **False**
4 $E^u_{\bar{T}} \leftarrow \emptyset$
5 **while** *finished* = **False do**
6 $t^u \leftarrow$ the last task in $\Lambda[u]$
7 $E^u_{\bar{T}}[t] \leftarrow \mathcal{T}(u, t^u, t).\varepsilon + \xi(u, t).\varepsilon \; \forall t \in \bar{T}$
8 $t \leftarrow \text{argmin}(E^u_{\bar{T}})$
9 **if** *remaining energy of* $u \geq E^u_{\bar{T}}[t]$ **then**
10 insert t into the end of $\Lambda[u]$
11 remove t from $\bar{T}$
12 put new available computation tasks into $\bar{T}$
13 **else**
14 *finished* $\leftarrow$ **True**
15 **if** $\bar{T} \neq \emptyset$ **then**
16 **return False**
17 **return True**

Next, we introduce Algorithm 2, which is the main algorithm that greedily maximizes the number of M-peered verification tasks, where $M \geq 2$ is an input. The algorithm works as follows:

1. Initialize the assignments of execution units with d_s (Line 1). Recall that d_s is the dummy start task introduced in Section 3.2.

2. Initialize the set of available tasks, $\bar{T}$, with the set of entry points in the given $\mathcal{G}$ (Line 2.)

3. Utilize NNH to pick the cheapest computation task t for an execution unit (Line 9), which is similar to what is done in Algorithm 1.

4. Greedily pick the cheapest peer to participate in the verification of t (Line 19). Check whether there are feasible assignments to the remaining tasks using Algorithm 1 (Line 21): If there is enough energy, we commit the peer (Line 22), update the assignments (Line 23), and try to add another peer. If there are no more peers to add or we have reached the maximum number of peers, go back to step 3.

3.4 Genetic Algorithm

A *genetic algorithm* (GA) requires a solution to the problem to be encoded by a string (aka. *chromosome*, *gene* or *individual*), and a set of chromosomes is referred to as a *population.* At initialization, a population is generated by some random method. Throughout the time, the population is evolved by mating, mutating, selecting its individuals. Upon reaching the terminating condition, the best individual in the population is selected as the final solution to the problem.

In this section, we present our proposed genetic algorithm (GA). The challenge in designing a GA to solve our problem is twofold: (1) verification tasks not only are optional but also require two or more execution units to start executing concurrently, and (2) the problem has two competing objectives which significantly reduce the convergence rate. To address these challenges, we first utilize a 3-part-string chromosome to encode a solution and propose the corresponding mating, mutating and selecting methods.

Algorithm 2: GreedilyVerification

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $\mathbb{U}$, M
Output: Λ `/* The set of assignments */`

$\Lambda[u] = \{d_s\}$, for $u \in \mathbb{U}$
$\bar{T} \leftarrow$ set of entry points in $\mathcal{G}$
while $\bar{T} \neq \emptyset$ **do**
  foreach $u \in \mathbb{U}$ **do**
    $t^u \leftarrow$ the last task in $\Lambda[u]$
    $E_{\bar{T}}[t] \leftarrow \mathcal{T}(u, t^u, t).\varepsilon + \xi(u, t).\varepsilon$, for $t \in \bar{T}$
    $t \leftarrow \text{argmin}(E_{\bar{T}})$
    if *remaining energy of* $u \geq \min(E_{\bar{T}})$ **then**
      insert t to the end of $\Lambda[u]$
      remove t from $\bar{T}$
      put new available computation tasks into $\bar{T}$
      break
  $\alpha_t \leftarrow$ the respective verification task of t
  $p \leftarrow \{u\}$
  $\tilde{\Lambda} \leftarrow \Lambda$
  append α_t to the end of $\tilde{\Lambda}[u]$
  while $|p| < M$ **do**
    $E_{\alpha_t}[u'] \leftarrow$ energy required by u' to participate in the verification, for $u' \in \mathbb{U} \setminus p$
    $u_p \leftarrow \text{argmin}(E_{\alpha_t})$
    append α_t to the end of $\tilde{\Lambda}[u_p]$
    if *GreedilyExplore*$(\mathcal{G}, \mathbb{U}, \tilde{\Lambda})$ = **True then**
      append u_p to p
      $\Lambda \leftarrow \tilde{\Lambda}$
    else
      break
return Λ

3.4.1 Representation of Solution

Each chromosome consists of:

- The *Selected Verification Tasks* (SVT) string contains $|\mathbb{U}|$ substrings. Each denoted by SVT_u, for each $u \in \mathbb{U}$, is a binary vector of length $|V_A|$, such that $SVT_u(i) = 1$ indicates that execution unit u participates in verification task α_i.

- The *Computation Task Assignment* (CTA) string is a vector of length $|V_T|$, such that $CTA(i) = j$, where $0 \leq i \leq |V_T|$, and $0 \leq j \leq |U|$; i.e., computation task t_i is assigned to unit u_j.

- The *Schedule String* (SS) is a vector of length $|\mathcal{V}|$, such that $SS(i) = (t, s_t)$, where $0 \leq i \leq |\mathcal{V}|, t \in \mathcal{V}$, and s_t is the start time of task t. Moreover, the vector is also a topological sort (Cormen *et al.*, 2009) of the verificable graph (i.e., the total ordering of tasks in the vector satisfies the precedence constraints.)

A chromosome is represented by a tuple $\langle SVT, CTA, SS \rangle$. Figure 3.1 illustrates a chromosome for the verificable task graph in Figure 2.2 and 2 execution units. For simplicity, we only show tasks in SS and omit their start times. In this example, (1) only verification tasks α_1, α_4, and α_6, are selected to be done by both execution units; (2) computation tasks t_2, t_4, and t_7 are assigned to execution unit u_2 while the others are assigned to u_1; and (3) the assignments for u_1 and u_2 are

$$\lambda_1 = (t_1, s_{t_1})(\alpha_1, s_{\alpha_1})(t_3, s_{t_3})(\alpha_4, s_{\alpha_4})(t_5, s_{t_5})(t_6, s_{t_6})(\alpha_6, s_{\alpha_6})$$

$$\lambda_2 = (\alpha_1, s_{\alpha_1})(t_2, s_{t_2})(t_4, s_{t_4})(\alpha_4, s_{\alpha_4})(\alpha_6, s_{\alpha_6})(t_7, s_{t_7})$$

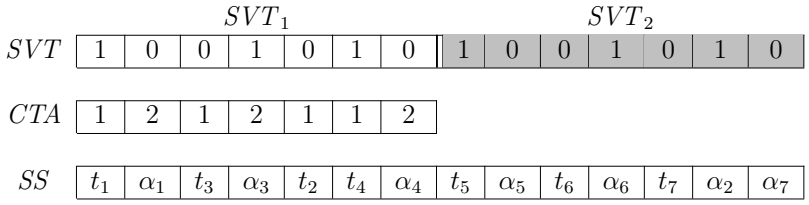


Figure 3.1: A sample chromosome.

3.4.2 Generation of Initial Population

Each part of a chromosome in the initial population is randomly generated as follows:

- **SVT**. First, a random binary string of length $|V_A|$ is generated for each execution unit $u \in \mathbb{U}$. Recall that $SVT_u(i) = 1$ means that execution unit u partakes in verification task α_i. When finishing the generation, only verification tasks done by 2 or more execution units will be retained. This enforces that SVT satisfies Constraint (3.1.8).

- **CTA**. First, computation tasks are randomly assigned to execution units wrt. their affinity. Next, we check every selected verification task if one of the execution units partaking in the verification task performs the respective computation task. If not, we randomly assign the computation task to one of the execution units wrt. their affinity. This enforcement makes the assignment partially satisfy Constraint (3.1.9), which additionally requires that an verification task must be executed right after its respective computation task. Moreover, CTA satisfies Constraints (3.1.4) and (3.1.5).

- **SS**. First, we utilize a modification of *Kahn's algorithm* (Kahn, 1962) to topologically sort the verificable graph and to enforce that an verification task must be done right after its corresponding computation task (Constraint (3.1.9)).

Another challenge in adapting the Kahn's algorithm is to concurrently assign multiple execution units to a single verification task which may create a deadlock (i.e., one execution unit waits for one other to authenticate computation tasks that they have just concurrently completed). For example, unit u_1 finished task t_1 and is waiting for u_2 to perform verification task α_1, while unit u_2 concurrently finished computation task t_2 and is also waiting for u_1 to perform verification task α_2. If such situation happens, we will cancel one of the verification tasks in the cycle and update the SVT. After obtaining the totally ordered sequence, we calculate the start time of each task and form the final SS.

Each newly generated chromosome is checked against those in the population. Two chromosomes are said to be the same if they have the same SVT, CTA, and sequence of tasks in SS. If a new chromosome is identical to any in the population, it is discarded, and the generation process is repeated. Otherwise, it will be put into the population. The rationale of obtaining only individual chromosomes is to increase both the diversity and convergence rate.

3.4.3 Crossover Operator

The crossover operator first randomly pick 2 chromosomes from the population and then processes their parts as follows:

- **SAT**. Recall that an SVT consists of $|\mathbb{U}|$ binary substrings of length $|V_A|$. The crossover operator randomly generates two cutoff points c_s (start point) and c_e (end point), such that $c_s, c_e \in [0, |V_A|]$ and $c_s < c_e$. The points divide each substring into three parts, and the middle part of each substring of a SVT will

be swapped with the respective middle part of the other SVT. Note that all substrings of a SVT are cut at the same points, so the new SVTs generated by the cross operator still satisfy the Constraint (3.1.8).

- **CTA**. We first apply a similar process as handling SVTs with two cutoff points. The swapping can never make the new individuals violate Constraints (3.1.4) and (3.1.5), but can violate Constraint (3.1.9). Therefore, we need to post-process the CTA to ensure that one of execution units partaking in an verification task does the corresponding computation task.

- **SS**. We simply invoke our Kahn algorithm, described at the end of Section 3.4.2, with the new SVT and CTA.

3.4.4 Mutate Operator

The mutate operator first arbitrarily selects a chromosome from the population and then processes its parts as follows:

- **SVT**. The mutate operator first randomly generates two cutoff points c_s and c_e as in the crossover operator to divide each substring into three parts. Next, it flips all the bits in the middle part. Finally, it keeps only verification tasks concurrently assigned to 2 or more execution units.

- **CTA**. We check every selected verification task α to enforce that one of the execution units partaking in α performs the respective computation task of α if necessary.

- **SS**. We simply invoke our Kahn algorithm, described at the end of Section 3.4.2, with the new SVT and CTA.

3.4.5 Selection

To identify which chromosomes are retained in or removed from the population, each is evaluated by:

- *Feasibility*, a Boolean value indicating if the chromosome represents a feasible solution or not. An individual is a feasible solution if it satisfies the energy bound.
- *CountSVT,* an integer whose absolute value is the number of verification tasks selected. CountSAT is a negative number if the chromosome is an infeasible solution.
- *CountPeers,* an integer whose absolute value is the total number of peers participating in all selected verification tasks. CountPeers is a negative number if the chromosome is an infeasible solution.

In order words, (1) each chromosome in the population is associated with a tuple $\langle feasibility, CountSVT, CountPeers \rangle$; (2) chromosomes are sorted in descending order wrt. the value of their associated tuples; and (3) only top P individuals will be selected to proceed to the next generation, where P is the size of the population.

3.5 Online Algorithm

While the offline algorithms are assumed to be theoretically correct (i.e., execution units should finish their assignments without running out of energy), the physical environment is likely to vary from the expectation set in the energy consumption parameters of tasks and execution units. For instance, weather conditions can affect

the duration and energy consumption of a UAV task, and humidity can affect the battery performance of a wireless sensor. Such changes may result in a reduction of peer-verification tasks, or more critically, impact the feasibility of the solution (see Constraint 3.1.5). We assume that a central entity controls execution units. If an execution unit detects a change in the environment, it informs the central controller which uses our proposed online algorithm to mitigate the change. We propose Algorithm 3 which takes as input an verificable task graph, a set of functional execution units, and a set of current assignments to the units. It operates in the following three stages.

3.5.1 Simulation

In the simulation phase (Line 3 of Algorithm 3), the algorithm simulates assignments $\tilde{\Lambda}$. Note that $\tilde{\Lambda}$ does not include assignments for execution units not in $\tilde{\mathbb{U}}$. Two scenarios may impact the simulation outcome: (1) a unit may run out of energy before finishing its assignment, and (2) peers participating in verification may be delayed due to longer than expected computation or transition. Once simulation is complete, we identify the set of computation tasks that were not completed by any execution unit in $\tilde{\mathbb{U}}$. If this is an empty set, our algorithm returns success indicating that the current set of assignments $\tilde{\Lambda}$ will result in a feasible solution. Otherwise, we denote the set of incomplete computation tasks as T' and proceed to the next step.

3.5.2 Backtracking

In this step, our objective is to identify verification tasks to eliminate from assignments $\tilde{\Lambda}$ to free up more energy for executing the remaining computation tasks in T'. We

Algorithm 3: RET online algorithm

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $\tilde{\mathbb{U}}$, $\tilde{\Lambda}$
Output: $\tilde{\Lambda}$

state $\leftarrow$ Infeasible
while *state = Infeasible* **do**
  $T' \leftarrow$ Simulate$(\mathcal{G}, \tilde{\mathbb{U}}, \tilde{\Lambda})$
  if $T' = \emptyset$ **then**
    state $\leftarrow$ Feasible
  else
    requiredEnergy $\leftarrow \sum_{t \in T'} \xi(u, t).\varepsilon$
    availableEnergy $\leftarrow 0$
    $V'_A \leftarrow \emptyset$
    while *availableEnergy* $<$ *requiredEnergy* **do**
      $\alpha \leftarrow$ the last unexecuted verification task in $\tilde{\Lambda}$
      if *there is no such* α **then**
        break
      $V'_A \leftarrow V'_A \cup \{\alpha\}$
      $U_\alpha \leftarrow$ set of units participating in doing α
      availableEnergy $\leftarrow$ availableEnergy $+|U_\alpha| \times \xi(u, \alpha).\varepsilon$
    if *availableEnergy* $<$ *requiredEnergy* **then**
      break
    $T' \leftarrow$ the set of completed tasks up to current time
    $A_{T'} \leftarrow$ the set of authenticate tasks whose respective computation tasks are in T'
    $\tilde{\mathcal{G}} \leftarrow \mathcal{G} \setminus \{V'_A \cup T' \cup A_{T'}\}$
    $\tilde{\Lambda} \leftarrow$GreedilyVerification$(\tilde{\mathcal{G}}, \tilde{\mathbb{U}}, M)$
if *state = Infeasible* **then**
  return Infeasible
else
  return $\tilde{\Lambda}$

propose a simple backtracking heuristic that works as follows:

1. Identify the minimum amount of energy required to execute the remaining tasks (Line 7). This is simply the sum of energy consumption of all tasks in T'. This is a lower bound on the actual energy required to execute the remaining tasks which could be higher due to transition costs. At this moment we do not know which transitions will be taken and so we assume all transitions cost zero energy.

2. Backtrack from the last verification task in $\tilde{\Lambda}$, adding up freed up energy until it is greater than or equal to the minimum amount of energy required to execute the remaining tasks (Line 16).

The output of this step is a set of verification tasks V'_A which will be removed from $\tilde{\Lambda}$. The current time refers to the time the central entity has received a notification of a change in the environment and invoked the online algorithm to reevaluate the current plan. If we cannot get enough freed up energy, this implies that there are no more verification tasks to remove to improve coverage of computation tasks, possibly because all other verification tasks have been executed already. In this case, we terminate the online algorithm with a failure due to infeasibility.

3.5.3 Calibration

In this step, we modify assignments $\tilde{\Lambda}$ to optimize for coverage of computation tasks by simply calling the greedy verification algorithm. In order to accelerate the algorithm, we invoke the algorithm on the modified graph which contains only unexecuted computation tasks and possibly executable verification tasks (Line 22).

3.6 Evaluation

To rigorously analyze the techniques proposed in this thesis, we target the multi-UAV exploration application discussed in Sections 1.1. That is, we employ set of UAVs (i.e., execution units) autonomously flying and doing tasks in a working area. We utilize the results of both simulations as well as *experiments on a real network of UAVs* to evaluate our algorithms. In this section, we start by presenting how we conduct simulations and experiments, then we compare the results in details.

3.6.1 Simulations

To do simulations, we first need to determine the service, transition and idle costs of execution units (i.e., UAVs) which is not trivial. To address this, we carried out a set of experiments on a Intel Aero Ready to Fly drone (Intel Aero RTF Drone, 2016) (see Figure 3.2).

That is, we let the drone do the jobs (i.e., scanning an area, flying from a place to another, staying idle for an amount of time) for a couple of dozens of times, measured the costs, and considered the mean values as the final costs. Note that we also applied the same strategy for simulations and experiments, which are represented latter (i.e., we repeated the simulation and experiments for couple of dozens of times, measured the costs, and considered the mean values as the final costs.)

We also model the working environment as a square grid. For example, Figure 3.3 illustrates a example of verificable task graph of a 2×3 grid. Each grid cell contains (1) a computation task (e.g., for scanning or detecting a certain type of object or anomaly), and (2) an verification task performed by two or more UAVs

Figure 3.2: Intel Aero Ready to Fly Drone.

(Formula (3.1.8)), which verifies the output of the computation task either by corroborating the result or authenticating peers. The gray circle denotes the depot from which all UAVs must take off and does not require a corresponding verification task (i.e., the depot is the dummy start task d_s introduced in Section 3.2). The solid arrows represent transitions based on task dependencies. That is, all UAVs must take off from the depot before flying to any cell, and the cells can be scanned in any order. The dashed lines represent the transitions from computation tasks to the corresponding verification task (Equation (2.2.1)).

It is evident that output of our algorithm is a *flight paths* for UAVs (i.e., execution units u) that accomplished a sequence of computation as well as verification tasks. Moreover, the assignment in $p_u^{\mathcal{S}}$ is identified such that the problem constraints identified in Section 3.1.3 are satisfied and the optimization objectives in Section 3.1.4 are

maximized. This, in turn, means that by solving our optimization problem, we will obtain a flight path for each UAV that maximizes the reliability of the output while staying within the energy bounds of the UAV's batteries.

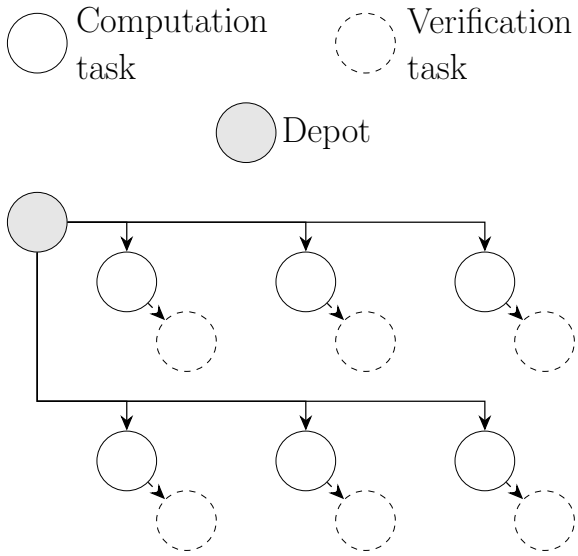


Figure 3.3: Verificable task graph of UAVs exploring a 2×3 grid map.

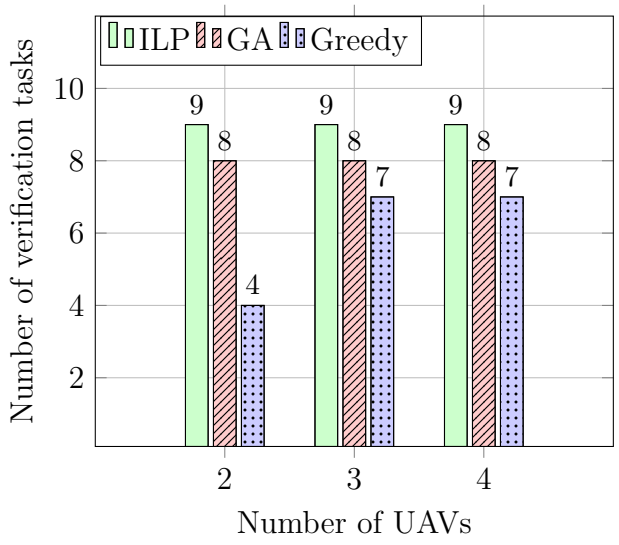
Moreover, there are also simulations for our online algorithm to evaluate how well the algorithms respond if one or more UAVs become unavailable during the execution. To this end, we created a multi-thread application in which one thread mimics the center entity, while the others mimic UAVs. While the application is running, we terminate a random UAV thread at an arbitrary time to simulate that the corresponding UAV becomes unavailable or crashes. The online algorithms are expected to be able to detect the issue and generate new flight paths for the remaining UAVs. Note that these simulations do not take into account changes in the physical environment which will be covered by the real experiments.

Analysis of Offline Algorithms

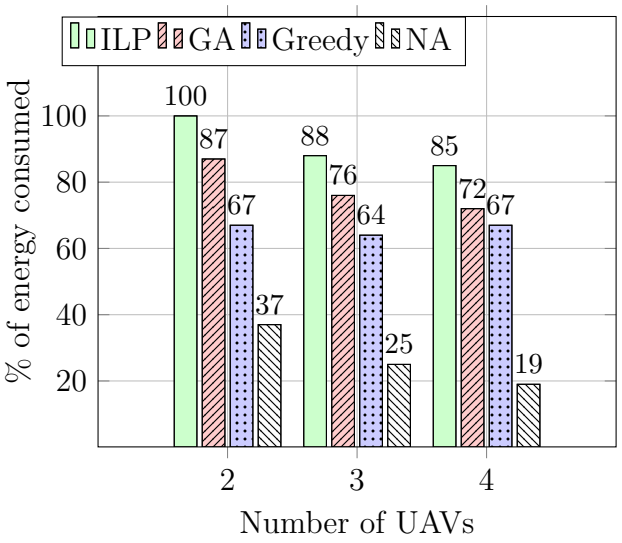
Figure 3.4 and 3.5 show the results of our simulations of the UAV system. We ran the ILP-based algorithm for the 3×3 grid to serve as a reference for what an optimal set of flight paths can achieve. For larger grids, ILP does not scale well. As expected, the ILP outperforms the genetic algorithm (GA) and the greedy algorithm in terms of number of verification tasks (see Figure 3.4a). The ILP also consumes more energy, utilizing the battery almost fully to maximize verifications.

Our simulations show that the GA consistently outperforms the greedy algorithm. The greedy algorithm always makes the cheapest choice which may not be beneficial in the long run. This is the heart of the problem that our work tackles: how verification decisions can non-immediately impact resource consumption and system objectives. Figure 3.4b shows that the GA utilizes the battery more efficiently. Note that NA refers to an algorithm that does not authenticate tasks at all. Figure 3.5b shows that there is a case where the greedy algorithm uses more energy than the genetic algorithm. This does not translate into more verifications (Figure 3.5a) since the greedy algorithm is making choices that incur more travel time costs in the future.

Moreover, the genetic algorithm is more inclined to schedule verifications at random points in time, further obscuring the verification process from an adversary. In the next section, we discuss simulation details of the online algorithm which is based on the genetic algorithm. Figures 3.4 and 3.5 demonstrate how verification tasks are randomly placed. This is opposed to the greedy algorithm that performs all verifications at the beginning.

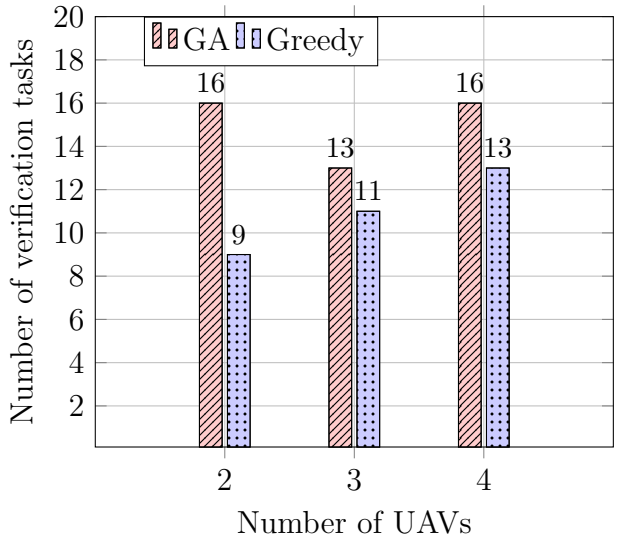


(a) 3×3 Verification

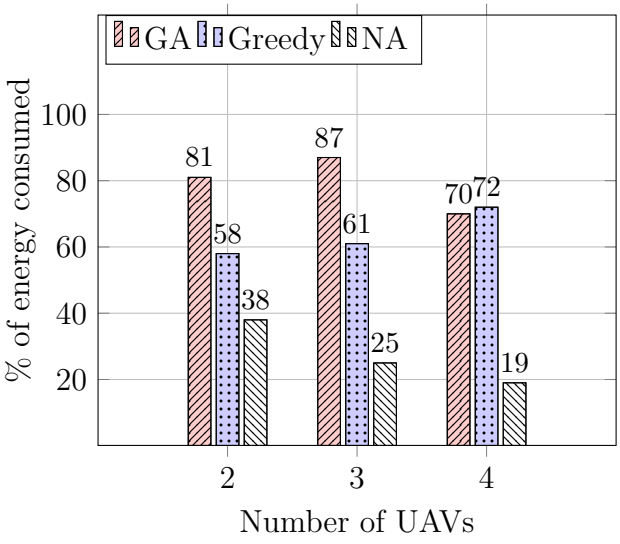


(b) 3×3 Energy Consumption

Figure 3.4: Simulation results for the RET problem on 3×3 grid.



(a) 4×4 Verification



(b) 4×4 Energy Consumption

Figure 3.5: Simulation results for the RET problem on 4×4 grid.

Analysis of the Online Algorithm

In order to analyze our online algorithm, we design the following scenarios. In the first scenario, four UAVs explore a 5×5 grid (i.e., our multi-thread application consists of five threads: one central entity and four UAVs). If the conditions during the entire operation do not change, then the UAVs complete their mission in $381s$ and accomplish 9 peer-verifications (see Figure3.6). This solution is obtained by the offline genetic algorithm. Now, we take the same flight paths and inject a UAV crash at time $170s$. The online algorithm realizes that the current flight paths are not feasible (due to reduction in the current energy bound) and computes a new schedule, which results in new flight paths and verification tasks for the remaining three UAVs. The new flight paths results in a total of 5 peer-verifications. In the third scenario, we inject an additional UAV crash at time $185s$. The drop in the total energy bound results in a situation where no further peer-verification can be achieved by the remaining to UAVs after the second crash.

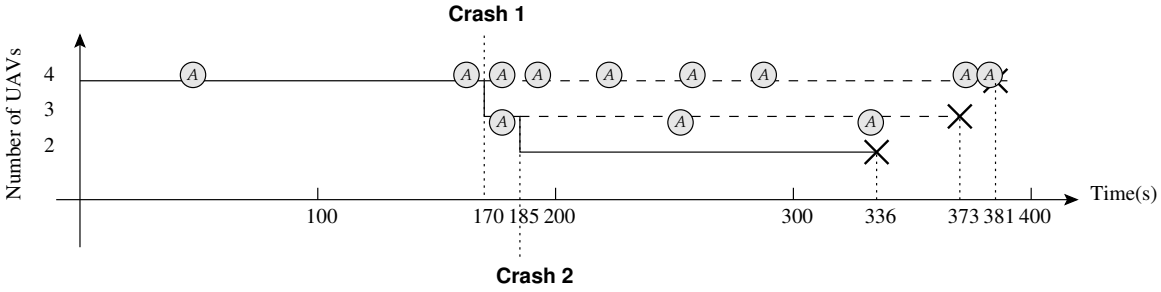


Figure 3.6: Simulation scenarios for the RET online algorithm.

3.6.2 Experiments with Real Multi-UAV Network

Our goal is to demonstrate that our online algorithm can react effectively to uncertainties caused by the physical environment. To this end, we employed two Intel

Aero Ready to Fly drones and programmed the PX4 flight controller to let them autonomously travel and do tasks in a 480-square-feet lab space which is modeled by a 4×4 square grid of (see Figure 3.7). Additionally, the UAVs communicate with each other and a ground station over a dedicated WiFi network. Note that, the UAVs originally utilize *global positioning system (GPS)* information to determine its current location and target. However, the GPS signal is not available indoor and is replaced by local position information provided by a motion capture system (12 OptiTrack motion capture cameras.) Note that for the safety reason, the two UAVs are flying at two different altitudes.

Moreover, the most challenge task in doing the experiment is to determine changes in costs at runtime which can only done by obtaining an energy model. To this end, we carried out a set of experiments (see Section 5.5.2), and utilize Polynomial Regression to derive the model.

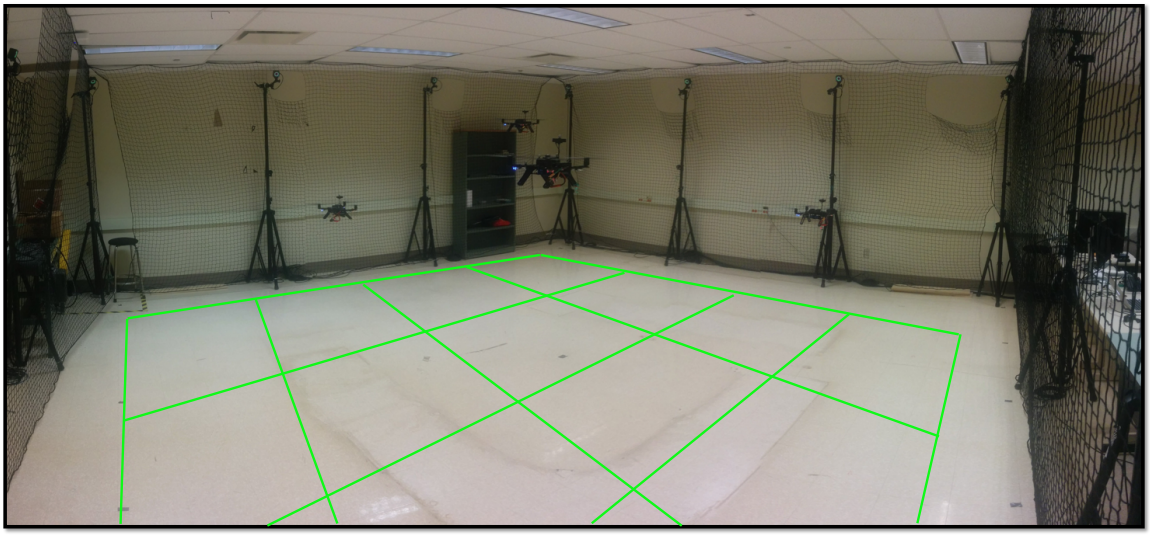


Figure 3.7: Experimental platform.

In experiments, the two UAVs (flying at two different altitudes) are supposed

to cover all 16 cells of the 4×4 grid. The genetic algorithm identifies flight paths with 16 computation tasks and 16 peer-verifications within the $200kJ$ battery energy bound of the UAVs. When the UAVs follow the flight paths prescribed by the genetic algorithm, they end up completing only 11 computation and 11 verification tasks due to reaching their energy limit too quickly, resulting in a pre-mature completion of the mission. The reason for this is that the top UAV creates spiral air flow, causing the lower UAV to spend about 12% more energy than predicted by the genetic algorithm.

Next, we ran our online algorithm initialized by the same flight paths as the genetic algorithm. The online algorithm effectively conducted appropriate re-planning and executed 16 computation tasks (i.e., covering the entire grid) and 9 verification tasks. That is, the online algorithm successfully managed the reliability-energy tradeoff within the energy limits of the UAVs.

Chapter 4

Optimal Recharging

In this chapter, we propose the *Optimal Recharging (OR)* problem which aims to extend the operational time of teams of battery-based robots by introducing a set of *charging stations* (a brief introduction of the problem was given in Section 1.2). We assume that the robots are heterogeneous (having different energy limits and being able to service different types of customers) and have access to a priori known map of the environment. The map is modeled as a directed, connected, and finite graph whose nodes are charging stations or customers, and arcs denote the possibility of traveling. Our ultimate goal is to find an optimal task assignment and path planning for the robots that minimizes energy consumption as well as the time needed to complete the tasks, including the time spent for recharging. To this end, we propose four *offline* optimization techniques and one *online* algorithm:

- First, we transform our optimization problem into *multi-objective integer programming* (MIP). This techniques produces the optimal result, though it is offline and limited to small problem sizes.

- Next, we introduce three scalable offline algorithms that find sub-optimal solutions. The first is a *semigreedy* algorithm which is a combination of the Nearest Neighbor Heuristic and an extension of the randomized $A^\star$ algorithm. We also propose two *genetic algorithms* (GAs) named *Greedy GA* and *random GA*. The two algorithms have the same evolving (mating, mutating and selecting) operations. For generating their initial population, the Greedy GA invokes the semigreedy algorithm, while the Random GA randomly assigns customers to robots.

- Finally, since offline solutions are often not suitable to deal with robots operating in real-life dynamic environments, we also propose an *online* algorithm to dynamically adjust the plan according to the changes in the physical environment.

We have fully implemented our algorithms and report results of simulation as well as experiments on a real network of UAVs. Our simulations show that while the MIP-based technique clearly identifies the optimal solution for small-size problems, for larger problems, our Greedy GA consistently outperforms both the semigreedy algorithm and Random GA. We also evaluate our online algorithm under scenarios, where a subset of the fleet of UAVs crash and the rest of the fleet has to adjust their plans accordingly and possibly recharge to cope with the new constraints. Our experiment shows how the online algorithm successfully distributes the remaining work to the functional UAV(s). We have also deployed our online algorithms on a real network of two UAVs that fly in two different altitudes and carry out a joint search mission.

The rest of this chapter is organized as follows. In Section 4.1, we formally state

the problem. Section 4.2 presents our transformation to MIP. Sections 4.3 and 4.4 introduce our offline heuristics, while Section 4.5 presents our online algorithm. Experimental results are analyzed in Section 4.6.

4.1 Problem Statement

In this section, we introduce the formal statement of the problem.

4.1.1 Cost Functions

To determine the costs of servicing customers in a route graph G for an execution unit $u \in \mathbb{U}$, we define the following cost functions:

- **Service cost** $\mathcal{S} : \mathbb{U} \times V_C \to \mathbb{R}^2_{\geq 0}$ is a function that maps an execution unit $u \in \mathbb{U}$ and a customer $c \in V_C$ to a pair of non-negative real numbers (τ, ε), where τ and ε are the amount of time and energy, respectively, consumed by u while servicing c. We also use $\mathcal{S}(u, c).\tau$ (respectively, $\mathcal{S}(u, c).\varepsilon$) to indicate the service time (respectively, energy).

- **Traveling cost** $\mathcal{T} : \mathbb{U} \times E \to \mathbb{R}^2_{\geq 0}$ is a function that maps an execution unit $u \in \mathbb{U}$ and an arc $a = (u, v) \in E$ to a pair of non-negative real numbers (τ, ε), where τ and *varepsilon* are the amount of time and energy consumed by u while traveling from u to v. We also use $\mathcal{T}(u, a).\tau$ (respectively, $\mathcal{T}(u, a).\varepsilon$) to indicate traveling time (respectively, energy).

- **Charging cost** $\mathcal{R} : \mathbb{U} \times V_B \to \mathbb{R}_{\geq 0}$ is a function that maps an execution unit $u \in \mathbb{U}$ and a charging station $b \in V_B$ to a non-negative real number τ denoting the amount of time required to fully recharge u at b.

4.1.2 Assumptions on Path Planning

First, we assume that:

- All execution units are initially fully charged to their capacity and located at the depot v_0.
- The depot and charging stations do not need to be serviced.
- For every pair of customers $v, v' \in V_C$, there is a path from v to v' without going through a charging station $b \in V_B$ or the depot v_0 (i.e., visiting the depot or a charging station is optional).

A *walk* of G is a sequence $w = v_0 v_1 \cdots v_n$, where v_0 is the depot (i.e., a walk must start from the depot), $v_i \in V$ for all $i \in [0, n]$, and $(v_i, v_{i+1}) \in E$ for all $i \in [0, n)$ (i.e., a walk must go through adjacent nodes in G.) We use $w(i)$ and $|w|$ to denote the ith node and the length of the walk, respectively. Note that, a node $v \in V$ can appear multiple times in w. We denote the set of all possible walks by W.

A *plan function* $\mathcal{F}$ maps an execution unit to a walk. Formally:

$$\mathcal{F} : \mathbb{U} \to W.$$

4.1.3 Problem Constraints and Optimization Objectives

First, let $w^u = v_0 \cdots v_n$ be a walk taken by an execution unit $u \in \mathbb{U}$. We denote the set of customers in w^u serviced by execution unit u by $C(w^u)$, that is:

$$C(w^u) = \big\{ w^u(i) \mid i \in [0, n] \wedge w^u(i) \in V_C \wedge w^u(i) \text{ is serviced by } u \big\}.$$

Also, let $B(w^u)$ be the *multiset* of charging stations in w^u visited by execution unit u, that is:

$$B(w^u) = \big\{ w^u(i) \mid i \in [0,n] \ \wedge \ w^u(i) \in V_B \big\}.$$

The total cost (time and energy consumption) of a walk w taken by an execution unit u (denoted $\mathbb{C}(w^u)$) is

$$\mathbb{C}(w^u) = \begin{bmatrix} \sum\limits_{c \in C(w^u)} \mathcal{S}\big(u,c\big).\tau + \sum\limits_{b \in B(w^u)} \mathcal{R}\big(u,b\big) + \sum\limits_{i=0}^{n-1} \mathcal{T}(u,(v_i,v_{i+1})).\tau \\ \sum\limits_{c \in C(w^u)} \mathcal{S}(u,c).\varepsilon + \sum\limits_{i=0}^{n-1} \mathcal{T}\big(u,(v_i,v_{i+1})\big).\varepsilon \end{bmatrix} \tag{4.1.1}$$

Recall that $B(w^u)$ is a multiset, which allows for multiple instances for each of its elements.

We now detail the constraints or our proposed problem. Our goal is to compute a plan function $\mathcal{F}$ subject to the following constraints.

Affinity. Every execution unit is only allowed to service customers as determined by its affinity set $\mathcal{A}_u$:

$$\forall u \in \mathbb{U} : C(\mathcal{F}(u)) \subseteq \mathcal{A}_u \tag{4.1.2}$$

where $C(\mathcal{F}(u))$ is the set of customers serviced by execution unit u when taking the walk determined by the plan $\mathcal{F}$.

Completion. All customers must be serviced exactly once.

$$\begin{gathered} \bigcup_{u \in \mathbb{U}} C(\mathcal{F}(u)) = V_C \\ C(\mathcal{F}(u)) \cap C(\mathcal{F}(u')) = \emptyset \quad \text{for each distinct } u, u' \in \mathbb{U} \end{gathered} \tag{4.1.3}$$

Energy bound. Each execution unit should respect its energy bound. For a walk

w^u, let $E_a(w^u, i)$ denote the remaining energy of u when it arrives at node $w(i)$, and $E_d(w^u, i)$ denote the remaining energy of u when it departs from node $w^u(i)$. The relations between E_a and E_d are as follows:

$$E_a(w^u, i+1) = E_d(w^u, i) - \mathcal{T}(u, (w(i), w(i+1))).\varepsilon \tag{4.1.4}$$

$$E_d(w^u, i) = \begin{cases} E_a(w^u, i) - \mathcal{S}(u, w^u(i)).\varepsilon \\ \qquad \text{if } w^u(i) \in C_u^w \wedge w^u(i) \text{ is serviced at } i \\ \mathcal{B}_u \qquad\qquad \text{if } w^u(i) \in B(w^u) \vee i = 0 \\ E_a(w^u, i) \qquad \text{otherwise} \end{cases} \tag{4.1.5}$$

where $i \in [0, n)$.

Recall that G is only connected (not complete) so an execution unit u may have to visit a node v several times but services it only once (if $v \in C(w^u)$.) Now, we constrain the energy consumption of execution unit u as follows:

$$\forall i \in [0, n] : E_a(w^u, i) \geq 0 \wedge E_d(w^{u,}, i) \geq 0 \tag{4.1.6}$$

Finally, our optimization objective is to to simultaneously minimize both time and energy consumption. That is:

$$min\left[\sum_{u \in \mathbb{U}} \mathbb{C}\big(\mathcal{F}(u)\big)\right]$$

4.2 Multi-objective Integer Program Transformations

In this section, we transform our problem into multi-objective integer programming (MIP). We start by identifying the challenges in the transformation:

- A route graph is incomplete. This means that an execution unit may have to travel through some nodes multiple times to reach its targets, which creates cycles in its walks and makes the transformation harder.
- To extend the operational time of execution units, we enforce them to visit charging stations before draining out their batteries, which is non-trivial. This also increases the possibility of having cycles in a walk.
- We intend to optimize both energy and time consumption to service all customers, including the time for recharging.

From the above challenges, it is clear that the desired MIP instance must have variables associated with the arcs in the given route graph G. Moreover, the variables must be integers to indicate how many times the arcs are used. However, this introduces a new challenge of deriving optimal walks for execution units from the solution of the MIP instance. For example, let consider the graph shown in Figure 4.1 where every arc has an associated number indicating how many times the arc is utilized. We can derive at least three different walks from the given graph: $w_1 = \cdots c_2bc_1bc_3bc_3c_4bc_4 \cdots$, $w_2 = \cdots c_2bc_3bc_1bc_3c_4bc_4 \cdots$ or $w_3 = \cdots c_2bc_4bc_1bc_3bc_3c_4 \cdots$.

Additionally, even if we can obtain the correct optimal walk, it is not trivial to determine when an execution unit should service a customer whom it visits more than

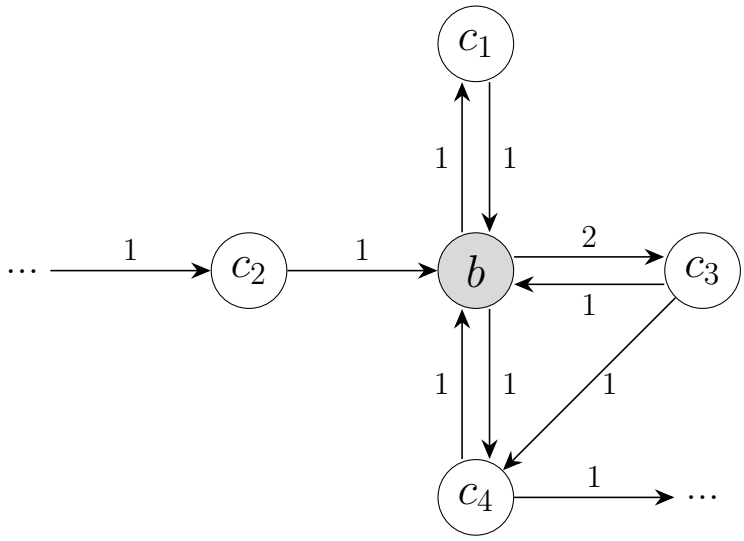


Figure 4.1: An example of a complex walk

once. For example, let us consider w_1 in which customer c_4 is visited twice (from c_3 and b), in which visit should an execution unit service c_4?

To completely resolve the above challenges, we transform the problem into two different MIP instances. The first one, presented in Section 4.2.1, is to identify the number of visits to each node. Then, in Section 4.2.2, we introduce the concept of *visit graph* and the second MIP instance to obtain optimal walks.

4.2.1 Obtaining Visits

Decision Variables

Let $\mathbb{U}$ be a finite set of execution units, $G = (V, E)$ be a route graph, and $\mathcal{S}, \mathcal{T}$ and $\mathcal{R}$ be the cost functions as defined in Section 4.1. The following are decision variables in our first MIP instance:

- $x^u_{(v,v')}$: non-negative integer variable indicating the number of times execution

unit u travels from v to v', for each $(v, v') \in E$.

- y_v^u: Boolean variable indicating if execution unit u services node v, where $v \in V_C$.

Constraints

This subsection details the constraints of the MIP instance based on the ones identified in Section 4.1.

Affinity. An execution unit can only service customers in its affinity set:

$$\sum_{v \notin \mathcal{A}_u} y_v^u = 0 \qquad \text{for all } u \in \mathbb{U}, v \in V_C \tag{4.2.1}$$

Walk validity. An execution unit

- must leave from the depot (v_0):

$$\sum_{(v_0,v) \in E} x_{(v_0,v)}^u \leq 1 \qquad \forall u \in \mathbb{U} \tag{4.2.2}$$

- goes through adjacent nodes:

$$0 \leq \sum_{(v,v') \in E} x_{(v,v')}^u - \sum_{(v',v'') \in E} x_{(v',v'')}^u \leq 1 \qquad \forall u \in \mathbb{U}, \forall v' \in V \setminus \{v_0\} \tag{4.2.3}$$

$$\delta\Big(\sum_{v \in W} y_v^u\Big) \cdot \Big(\sum_{(v,v') \in \alpha(W)} x_{(v,v')}^u - 1\Big) \geq 0 \qquad \forall u \in \mathbb{U}, \forall W \subseteq V \setminus \{v_0\} : V_C \cap W \neq \emptyset \tag{4.2.4}$$

where $\alpha(W) = \{(v_i, v_j) \,|\, (v_i, v_j) \in E \wedge v_i \notin W \wedge v_j \in W\}$ (i.e., the set of arcs that have only head ends in W,) and $\delta(x)$ is the Kronecker delta function, which returns 0 if $x = 0$, otherwise 1. Equation (4.2.3) is the *flow constraint* enforcing that execution unit u can only leave a node only after having arrived it. Equation (4.2.4) is called *connectivity constraint* and is adapted from the one proposed by Fleishmann in (Fleischmann, 1985). In our adaption, the delta function is utilized to cover both situations whether execution unit u services any customers in W or not. In the former case, $\delta(\sum_{v \in W} y_v^u)$ returns 1 enforcing that execution unit u has to take at least an arc in $\alpha(W)$ to go into W and service the customers. In the latter case, $\delta(\sum_{v \in W} y_v^u)$ returns 0 making the constraint tautology regardless of value of x's. Furthermore, Equation (4.2.4) makes our instance *non-linear* and requires a *non-linear multi-objective solver.*

- does not return to the depot:

$$\sum_{(v,v_0) \in E} x^u_{(v,v_0)} = 0 \qquad \forall u \in \mathbb{U} \tag{4.2.5}$$

Completion. All customers must be serviced exactly once:

$$\sum_{u \in \mathbb{U}} y_v^u = 1 \qquad \forall v \in V_C \tag{4.2.6}$$

The binary variable y_v^u depends on $x^u_{(v',v)}$ as follows:

$$\sum_{(v',v) \in E} x^u_{(v',v)} \geq y_v^u \qquad \forall u \in \mathbb{U}, \forall v \in V_C \tag{4.2.7}$$

That is, if execution unit u services customer v, then it must travel to v.

Energy bound. Every execution unit must operate within its energy bound. To constrain energy consumption, we first calculate the amount of energy required by execution unit u to travel and service a subset of customers W:

$$\gamma^u(W) = \sum_{v \in W} y_v^u \cdot \mathcal{S}(u,v).\varepsilon + \sum_{(v,v') \in \beta(W)} x^u_{(v,v')} \cdot \mathcal{T}(u,(v,v')).\varepsilon$$
$$\forall W \subseteq V_C$$

where $\beta(W) = \{(v,v') \mid (v,v') \in E \wedge v,v' \in W\}$ (i.e., the set of arcs that have both ends in W). Next, $\zeta^u(W)$ is the amount of energy spent by execution unit u to travel in and out a subset of customers W:

$$\zeta^u(W) = \sum_{(v,v') \in \alpha(W)} x^u_{(v,v')} \cdot \mathcal{T}(u,(v,v')).\varepsilon + \sum_{(v,v') \in \omega(W)} x^u_{(v,v')} \cdot \mathcal{T}(u,(v,v')).\varepsilon$$
$$\forall W \subseteq V_C$$

where $\omega(W) = \{(v,v') | (v,v') \in E \wedge v \in W \wedge v' \notin W\}$(i.e., the set of arcs that have only tail ends in W). Finally, we constrain the energy consumption of execution unit u as follows:

$$\delta\Big(\sum_{v \in W} y_v^u\Big) \cdot \Big(\sum_{(v,v') \in \alpha(W)} x^u_{(v,v')} - \frac{(\gamma^u(W) + \zeta^u(W))}{\mathcal{B}_u}\Big) \geq 0 \qquad (4.2.8)$$
$$\forall W \subseteq V_C$$

Equation (4.2.8) is an adaption of Equation (4.2.4). Again, the delta function is utilized to cover the both situations (whether execution unit u services any customers in W or not.) In the latter case, the constraint is tautology and can be ignored. In

the former case, the rightmost fraction returns the minimum number of times that u has to be recharged. Thus, the combination of the equation with Equations (4.2.3), (4.2.4), and (4.2.5) ensures that u will go out of W to a charging station and return to W to continue its assignment when necessary.

Objective Function

As mentioned in Section 4.1, our problem requires optimizing both energy and time consumption. Let $E_B = \{(v, v') | (v, v') \in E \wedge v' \in V_B\}$ (i.e., the set of arcs whose head ends are recharge stations). Equation (4.2.9) is our objective function.

$$\min \left[\begin{array}{l} \sum_{u \in \mathbb{U}} \Big[\sum_{(v_i, v_j) \in E} x^u_{(v,v')} \cdot \mathcal{T}(u, (v, v')).\tau + \sum_{(v,v') \in E_B} x^u_{(v,v')} \cdot \mathcal{R}(u, v') + \sum_{v \in V_C} y^u_v \cdot \mathcal{S}(u, v).\tau \Big] \\ \sum_{u \in \mathbb{U}} \Big[\sum_{(v,v') \in E} x^u_{(v,v')} \cdot \mathcal{T}(u, (v, v')).\varepsilon + \sum_{v \in V_C} y^u_v \cdot \mathcal{S}(u, v).\varepsilon \Big] \end{array} \right] \tag{4.2.9}$$

4.2.2 Obtaining Optimal Walks

Visit graph

Let $S^\star = (V^\star, E^\star)$ be the directed graph encoding the optimal solution obtained from the MIP proposed in Section 4.2.1, where:

$$E^\star = \Big\{ (v, v') \mid (v, v') \in E \wedge \sum_{u \in \mathbb{U}} x^u_{(v,v')} > 0 \Big\}$$

$$V^\star = \Big\{ v, v' \mid (v, v') \in E^\star \Big\}$$

We associate each $v \in V^\star$ with a set of visits $V_v^\star = \{v^1, \cdots, v^{m_v}\}$, where $m_v = \sum_{u \in \mathbb{U}} \sum_{(v',v) \in E^\star} x^u_{(v',v)}$. Now, let

$$\mathcal{V}_\mathcal{C} = \bigcup_{v \in V_C} V_v^\star$$
$$\mathcal{V}_\mathcal{B} = \bigcup_{v \in V^\star \cap V_B} V_v^\star$$
$$\mathcal{V} = \mathcal{V}_\mathcal{C} \cup \mathcal{V}_\mathcal{B} \cup \{v_0\}$$

That is, the set of vertices $\mathcal{V}$ includes all visits to the customers and the charging stations. Note that, unlike customers (all of which must be serviced and included in $V^\star$), there could exist charging stations that are never visited by any execution unit in the optimal solution $S^\star$, and they will not be included in $\mathcal{V}_\mathcal{B}$. Finally, the corresponding visit graph is $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where the set of directed edges $\mathcal{E}$ are as follows:

$$\mathcal{E} = \left\{(v_i^k, v_j^l) \mid (v_i, v_j) \in E^\star \wedge v_i^k \in V_{v_i}^\star \wedge v_j^l \in V_{v_j}^\star\right\} \tag{4.2.10}$$

Example Figure 4.2 illustrates the visit graph of the problem in Figure 2.3 where customers c_1, c_2, and c_6 are visited twice. As can be seen in the figure, we introduce three more nodes c_1^2, c_2^2, and c_6^2 which denotes the second visits to the customers, respectively. We also include additional edges $((v_0, c_1^2), (c_1^2, c_2^1), \cdots)$ to retain the corresponding transition-ability between the nodes (see Equation (4.2.10).)

The second MIP instance

The input parameters for this transformation are (1) the visit graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, which is constructed from the result of the first MIP instance, (2) the set of execution units

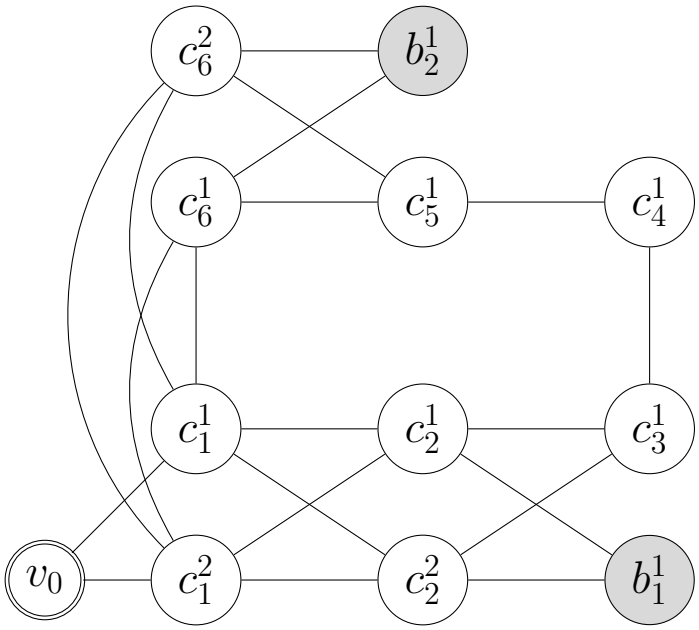


Figure 4.2: An sample of visit graph

$\mathbb{U}$, (3) and the cost functions $\mathcal{S}, \mathcal{T}$ and $\mathcal{R}$. Again, we utilize the two type of decision variables as in previous transformation but they are both Booleans. That is:

- $x^u_{(v,v')}$: Boolean variable indicating if execution unit u visits v' right after v, where $(v, v') \in \mathcal{E}$.

- y^u_v: Boolean variable indicating if execution unit u services v, where $v_i \in \mathcal{V}_\mathcal{C}$.

Next, the objective function (4.2.9), and constraints (4.2.1), (4.2.2), (4.2.3), (4.2.4), (4.2.5), (4.2.7) and (4.2.8) are reused after replacing V, E, V_C and V_B by $\mathcal{V}, \mathcal{E}, \mathcal{V}_\mathcal{C}$ and $\mathcal{V}_\mathcal{B}$, respectively. Additionally, we introduce Constraints (4.2.11) and (4.2.12) to enforce that each node in $\mathcal{G}$ is visited only once, and each customer is serviced only once, respectively.

$$\sum_{(v,v')\in\mathcal{E}} x^u_{(v,v')} = 1 \qquad \forall u \in \mathbb{U}, v' \neq v_0 \tag{4.2.11}$$

$$\sum_{u\in\mathbb{U}} \sum_{v\in V^\star_{v'}} y^u_v = 1 \qquad \forall v' \in V_C \tag{4.2.12}$$

4.3 Semigreedy Algorithm

In this section, we introduce our semigreedy algorithm (Algorithm 5) using Nearest Neighbor Heuristic (NNH) (Bellmore and Nemhauser, 1968). The NNH utilizes a loop to sequentially select a customer which has not been assigned to any execution units and has the lowest insertion cost to assign to an execution unit. Recall that the insertion cost of a customer includes both the traveling and servicing cost required to append the customer to the end of the partially build walk of an execution unit. Moreover, we also add randomization into the selection procedure making the ordinary NNH into a semigreedy heuristic version. Before discussing the detail of the Algorithm 5, let us first identify three specific properties that hold in every optimal subwalk $w_u^{(i,j)} = (v_i, \cdots, v_j)$ assigned to execution unit u, such that v_i and v_j are *the only two customers serviced by* u while the other nodes between them (if any) are intermediate ones (i.e., due to the incompleteness of route graph, u may have to visit intermediate nodes in order to reach v_j). We use these properties to construct subwalks in Algorithm 4, which is used by our semigreedy Algorithm 5:

- If execution unit u does not visit any charging station while executing $w_u^{(i,j)}$, then it will visit every intermediate node once. This is true because if there exists an intermediate node v which is visited more than once, then the subwalk has

one ore more cycles which are associated with v and can be removed to obtain a better subwalk which is a contradiction.

- If execution unit u visits a charging station v_b while executing $w_u^{(i,j)}$, then v_b is visited only once. This is true because if v_b is visited more than once, then there exists one or more cycles which are associated with v_b and can obviously be removed to get a better subwalk which is a contradiction.

- If execution unit u has to visit one or more charging stations while executing $w_u^{(i,j)}$, and let v_b be the very first charging station that execution unit u visits after v_i, then every intermediate node v between v_i and v_b is visited only once. This property also holds when v_i is the depot or a charging station. This is true because if there exists an intermediate node v which is between v_i and v_b and is visited more than once, then there exists one or more cycles which are associated with v and can be removed to obtain a better subwalk which is a contradiction.

Therefore, it is possible to partition $w_u^{(i,j)}$ into small *(simple) paths* (i.e., a sequence does not include any repeated node) starting from v_i to the very first charging station, from this to the very next charging station, so on so forth until to v_j.

Next, we explore a slight adaptation of the Multi-Objective $A^\star$ (MOA*) algorithm proposed in (Mandow *et al.*, 2005) to find all shortest paths between two nodes. The original algorithm takes as input a graph, a start node and a set of goal nodes, and returns a directed acyclic graph (DAG) encoding all shortest paths between the start and each goal. Moreover, MOA*, like any other variant of $A^\star$ algorithm (Hart *et al.*, 1968), requires a heuristic function $\mathcal{H} : U \times V \times V \to \mathbb{R}^2_{\geq 0}$ that maps an execution

unit u and 2 vertices $v, v' \in V$ to a pair of real numbers (t, e), where t and e are the estimate time and energy consumed by execution unit u to travel from v to v'. In this paper, we make the following changes to the MOA*:

- The set of goal nodes is replaced by a single target node v_t.
- It takes two additional parameters: execution unit u and the amount of energy r that u should reserve when reaching v_t. Execution unit u will utilize this energy to service v_t and move to the closest charging station (if necessary).
- In the *path expansion* procedure, we add steps to update the remaining energy of execution unit u according to Equations (4.1.4) and (4.1.5), and only neighbors which do not violate Equation (4.1.6) (the energy constraint) will be selected for further exploration.

Now, we discuss Algorithm 4, which constructs the aforementioned subwalk $w_u^{(i,j)}$ (i.e., v_i and v_j are the only two customers serviced by u, and the other nodes between them (if any) are intermediate ones.) The algorithm takes as input the route graph G, an execution unit u, a start node v_i, a target node v_j, and an amount of energy r that u should reserve, and returns a shortest subwalk w between the two nodes. It works as follows:

1. Initialize w by v_i (Line 1).
2. Initialize the stack of final and intermediate stops (i.e., charging stations) with v_j (Line 2).
3. Utilize a loop (Lines 4 - 19) to construct paths. At each iteration,

Algorithm 4: ConstructSubwalk

Input: G, u, v_i, v_j, r
Output: a shortest walk W from v_i to v_j

```
1  w ← [v_i]
2  stops ← [v_j]
3  s ← v_i
4  while True do
5      t ← pop the last node from stops
6      r' ← 0
7      if t = v_j then
8          r' ← r
9      Paths ← MOA*(G, u, s, t, r')
10     if Paths ≠ ∅ then
11         p ← randomly pick one in Paths
12         extend w with p
13         if t = v_j then
14             return w
15         s ← t
16     else
17         push t back to stops
18         greedily choose an unvisited charging station b between s and t
19         push b to the end of stops
```

(a) invoke MOA* to find a shortest path between the corresponding start and target nodes (Line 9),

(b) if there exists a path (Lines 10 - 15), we extend the current walk with the newly found path and terminate if reaching the final destination,

(c) Otherwise (Lines 16 - 19), we utilize the heuristic function $\mathcal{H}$ to greedily find a charging station, which is the nearest one between the two current ends and has not been visited, and push to the stack of intermediate stops. Note that as shown earlier, every charging station can be visited only once in subwalk $w_u^{(i,j)}$ so if there does not exist such station, the algorithm will

Algorithm 5: OR semigreedy algorithm

Input: G, $\mathbb{U}$
Output: Walks for the execution units

```
1  Ṽ_C ← V_C
2  W̃[u] ← [v_0], ∀u ∈ 𝕌
3  while Ṽ_C ≠ ∅ do
4      foreach (u, c) ∈ 𝕌 × Ṽ_C do
5          r ← S(u, c)
6          if c is not the last customer then
7              e ← min{H(u, c, b)}, for all b ∈ V_B
8              r ← S(u, c) + e
9          l ← the last node in W̃[u]
10         w ← ConstructSubwalk(G, u, l, c, r)
11         InsertCosts [u, c] = C(w^u) + S(u, c)
12     S ← argmin(InsertCosts)
13     (u, c) ← randomly pick one from S
14     update W̃[u]
15     remove c from Ṽ_C
16 return W̃
```

terminate immediately (i.e., it cannot find any path between v_i and v_j without violating energy constraints.)

Finally, we introduce Algorithm 5 that greedily solves the proposed problem. It takes as input a route graph G and the set of execution units $\mathbb{U}$. The algorithm works as follows:

1. Initialize the set of unserviced customers with all customers (Line 1).

2. Initialize the partially built walk of each execution units with the depot (Line 2).

3. Utilize a loop (Lines 3 - 15) to sequentially assign an unserviced customer to an execution unit. At each iteration,

(a) Determine the amount of energy that an execution unit has to reserve after reaching a new customer (Lines 5 - 8). If there are two or more unserviced customers, an execution unit has to reserve energy not only to service the selected customer but also to move to the nearest charging station after servicing the customer. By doing so, we can get rid of the premature completion (i.e., although there still exists unserviced customers, all execution units have completely or nearly drained out their batteries at their last customers and cannot go to any charging stations.)

(b) Utilize Algorithm 4 to determine a shortest subwalk to the next customer (Line 10), and calculate the corresponding insertion cost (Line 11). Recall that, $\mathbb{C}(w^u)$ returns total cost of a walk w taken by execution unit u (see Equation 4.1.1).

(c) Due to the competing of two objectives, there likely exists more than one pair of a customer and an execution unit whose insertion costs are not (Pareto) dominated. Let S be the set of such pairs (Line 12). Recall that given two vectors $y_1 = (x_1^1, x_2^1, \cdots, x_n^1)$ and $y_2 = (x_1^2, x_2^2, \cdots, x_n^2)$, y_1 is said to (Pareto) dominate y_2 (denoted by $y_1 \prec y_2$) if and only if $\forall i \in [1, n], x_i^1 \leq x_i^2$ (Deb, 2014).

(d) Randomly pick a pair from the set S, and update the corresponding partially built walk of the selected execution unit. Note that, we does not only append the selected customer to the end of the walk but also the intermediate nodes. This randomization step makes the greedy algorithm into a semigreedy algorithm.

(e) Update the set of unserviced customers by removing the newly selected

customer.

4.4 Genetic Algorithm

4.4.1 Representation of Solution

Recall that, our problem has two competing objectives (optimizing both time and energy consumption) which results in the existence of more than one shortest subwalks between any two customers v_i and v_j. In other words, given a same set of customers assigned to an execution unit, we can have multiple shortest walks which could be assigned to an execution unit. In order to include all such walks in our population, we utilize a 2-part string to encode an individual:

- The *Customer Assignment String* (CAS) is a vector of length $|V_C|$, such that $CAS(i) = j$, where $1 \leq i \leq |V_C|$ and $1 \leq j \leq |\mathbb{U}|$ are IDs of a customer and an execution unit, respectively. That is, customer $c_i \in V_C$ is assigned to execution unit $u_j \in \mathbb{U}$.

- The *Walks String* (WS) contains $|\mathbb{U}|$ substrings each is denoted by WS_u, for each $u \in \mathbb{U}$, and represents a walk executed by the associated execution unit.

Each individual is represented by a pair $\langle CAS, WS \rangle$. Figure 4.3 illustrates a chromosome for the problem in Figure 2.3 and two execution units. In this example, (1) customers c_1, c_2, and c_3 are assigned to execution unit u_1 while customers c_4, c_5,

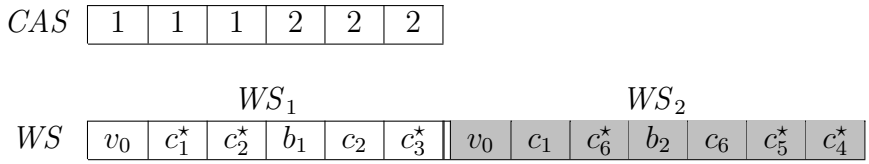


Figure 4.3: A sample individual

and c_6 are assigned to execution unit u_2; (2) and the walks for u_1 and u_2 are:

$$w_{u_1} = [v_0, c_1^\star, c_2^\star, b_1, c_2, c_3^\star]$$

$$w_{u_2} = [v_0, c_1, c_6^\star, b_2, c_6, c_5^\star, c_4^\star]$$

Note that, we utilize $\star$ to indicate the visit during which the execution unit services the corresponding customer.

Moreover, we want only distinctive individuals in our population. To this end, each newly created chromosome (by random generation, crossover operation, or mutation) is checked against those in the population. If the new chromosome is identical to any in the population, it is discarded. Otherwise, it will be put into the population. Two chromosomes are said to be the same if they have the same *CAS* and *WS*. The rationale of obtaining only distinctive individuals is to increase both the diversity and convergence rate.

4.4.2 Generation of Initial Population

Any GA algorithm requires a set of initial individuals from which the evolution begins. However, our intuition is that the initial population can give an important contribution to enhance the final solution. To test this, we utilize two different methods for

generating the initial population and compare their final results. In the first method, named *Greedy GA*, we invoke Algorithm 5 to obtain customers and walks assigned to each execution unit (i.e., *CAS* and *WS*, respectively). In the second method, named *Random GA*, we first generate the *CAS* part by randomly assigning a customer to an execution unit (that is, we generate a random integer number in the range of $[1, |\mathbb{U}|]$ and assign it to a $CAS(i)$ which represents customer $v_i \in V_C$). Then, we construct a walk for every execution unit to visit its customers using Algorithm 6, which takes as input the problem graph G, an execution unit u and the set of customers assigned to u. The algorithm works as follows:

1. Initialize the set of unserviced customers with V_C^u (Line 1).
2. Initialize the walk with the depot v_0 (Line 2).
3. In the loop (Lines 3 - 15), utilize NNH to sequentially assign each customer to the unit. At each iteration,
   (a) evaluate the insertion cost of each customer (Lines 4 - 12),
   (b) pick a customer with the lowest insertion cost (Line 13). If there exists more than one (Pareto) nondominate customers, then randomly pick one of them,
   (c) update the currently built walk of the execution unit. Note that, we do not simply append the selected customer to the end of the walk, but also append the intermediate nodes (if any),
   (d) remove the newly selected customer from the set of unserviced ones.

Algorithm 6: ConstructWalk

Input: G, u, V_C^u
Output: Walk for execution unit u

$\tilde{V_C} \leftarrow V_C^u$
$w \leftarrow [v_0]$
while $\tilde{V_C} \neq \emptyset$ **do**
    InsertCosts $\leftarrow []$
    $l \leftarrow$ the last node in w
    foreach $c \in \tilde{V_C}$ **do**
        $r \leftarrow \mathcal{S}(u, c)$
        if *c is not the last customer* **then**
            $e \leftarrow \min\{\mathcal{H}(u, c, b)\}$, for all $b \in V_B$
            $r \leftarrow \mathcal{S}(u, c) + e$
        $p \leftarrow ConstructSubwalk(G, u, l, c, r)$
        InsertCosts[c] $\leftarrow \mathbb{C}(p^u) + \mathcal{S}(u, c)$
    $c \leftarrow \text{argmin}(InsertCosts)$
    update w
    remove c from $\tilde{V_C}$
return w

4.4.3 Crossover Operator

Recall that an individual is a pair $\langle CAS, WS \rangle$, where CAS is the major component encoding the set of customers assigned to an execution unit, and WS is the set of walks for execution units. Our proposed crossover operator mainly targets to CAS component and reproduces the corresponding WS. The detailed procedure is as follows:

1. Randomly pick two individuals from the population.

2. Generate two cutoff points s (start point) and e (end point), such that $s, e \in [1, |V_C|]$ and $s < e$.

3. Utilize the points to divide the CAS of each individual into three pieces.

4. Swap the middle piece of each individual.

5. Invoke Algorithm 6 to generate new WS and replace the existing one.

4.4.4 Mutate Operator

Mutation of a individual is implemented through a very simple procedure:

1. Randomly choose an individual.

2. Generate two cutoff points s (start point) and e (end point) for the CAS as in crossover operator.

3. Apply a circular shift to the middle part. That is, replace $CAS[s]$ with $CAS[s+1]$, $CAS[s+1]$ with $CAS[s+2]$, ..., $CAS[e]$ with $CAS[s]$.

4. Invoke Algorithm 6 to update the WS.

4.4.5 Select Operator

New individuals are introduced by crossover and mutation operators at every generation, and in order to maintain a fixed-size population, we have to remove some bad individuals before proceeding to the next generation. To this end, first, each in the population is associated with a pair including:

- *Infeasibility*, a Boolean value indicating if any execution units of the solution violate the energy constraints (see Equation 4.1.6).

- *Total cost*, a pair of numbers indicating the total time and energy consumption of all execution units.

Then, all individuals in the population are sorted in ascending order wrt. the value of their associated pair. Finally, we keep only top P individuals in the sorted population, where P is the size of the population.

4.5 Online Algorithm

In the context of DAMR systems, the physical environment (e.g., weather conditions) likely affects the parameters set in our offline algorithms. Such changes motivate the need of an online algorithm which adjusts walks assigned to execution units when necessary. We assume that (1) a central ground station (GS) controls execution units, and (2) only traveling cost, servicing cost, and the set of execution units change over the time. That is, while the depot, the set of customers, charging stations and route graph remain unchanged. Let $\mathcal{T}_{on}$ and $\mathcal{S}_{on}$ denote the online traveling and servicing cost functions.

Next, we discuss the input of the proposed algorithm. First, when the GS receives a notification of changes in the environment at time t, there could be some (or all) customers which have been serviced and must be eliminated before invoking the algorithm. Similarly, only functional execution units $\tilde{\mathbb{U}}$ at time t are taken as input of the online algorithm. Next, let us consider a walk $w = v_0 v_1 \cdots v_n$ assigned to execution unit u. If u locates at v_i at the time t (if it is traveling to v_i, we still consider that u is at v_i), where $i \in [0, n]$, then the *partially executed walk* of u is $\tilde{w} = [v_i \cdots, v_n]$. Let $\tilde{W}$ be the set of all partially executed walks of functional execution units and be the last input of our proposed algorithm. Note that, on eliminating a serviced customer, we do not truly remove it from G but only mark it as an intermediate node (recall that our graph G is incomplete and we may need it to reach unserviced customers.) In

other words, the vertices of G at time t can be separated into 3 disjoint sets: the set of charging stations V_B, the set of unserviced customers $\tilde{V_C}$, and the set of intermediate nodes V_I. We denote the graph modeling the problem at time t by $\tilde{G}$. Finally, we introduce Algorithm 7 which takes as input a graph $\tilde{G}$, a set of functional execution units $\tilde{\mathbb{U}}$, and a set of partially executed walks $\tilde{W}$. Our proposed online algorithm operates in the following two phases:

4.5.1 Simulation

The major purpose of this phase (Line 1) is to identify the set of customers $\bar{V_C}$ which newly become unassigned because one or more execution units become unavailable (e.g., because of crashes) or violate energy bound (i.e., the changes in the physical environment make the execution units consume more energy than expected and be unable to arrive a charging station before draining out their batteries.) To detect the violation in the latter case, we utilize the online cost functions (i.e., $\mathcal{T}_{on}$ and $\mathcal{S}_{on}$) to check the remaining energy of each execution unit at every node in the assignments (see Equation 4.1.6).

4.5.2 Calibration

In this phase, we introduce two optimizing strategies.

Locally optimizing. This strategy is applied when $\bar{V_C}$ is an empty set (i.e., there does not exist any unassigned customer) (Lines 2 - 10). Although the current walks can be still utilized, we can get a better solution by targeting intermediate nodes. That is, if execution unit u has to travel through an intermediate node v which is

also an unserviced customer, we try to assign v to u. If the assignment yields a better result, then it is committed.

Resolving. This strategy is to assign customers in $\bar{V_C}$ to functional execution units (Lines 11 - 19). To this end, we apply NNH to assign each customer in $\bar{V_C}$ to an execution unit u which has the lowest insertion cost. Recall that, $\mathbb{C}(w^u)$ is the cost of walk w assigned to execution unit u (see Equation 4.1.1.)

Algorithm 7: OR online algorithm

```
   Input: G̃, 𝕌̃, W̃
   Output: new walks for functional execution units 𝕌̃
 1 V̄_C ← Simulate(G̃, 𝕌̃, W̃)
 2 if V̄_C = ∅ then
 3     Ṽ_C ← set of unserviced customers
 4     while Ṽ_C ≠ ∅ do
 5         foreach u ∈ 𝕌̃ do
 6             I_u ← the set of intermediate node in W̃[u]
 7             foreach v ∈ {v' | v' ∈ I_u, v' ∈ V̄_C} do
 8                 if assigning v to u yields a better solution then
 9                     assign v to u and update W̃[u]
10                     remove v from Ṽ_C
11 else
12     foreach c ∈ V̄_C do
13         foreach u ∈ 𝕌̃ do
14             C_u ← set of customers assigned to u
15             C_u ← C_u ∪ {c}
16             w_u ← ConstructWalk(G, u, C_u)
17             Costs[u] ← ℂ(w^u)
18         u ← argmin(Costs)
19         assign c to u and update W̃[u]
20 return W̃
```

4.6 Evaluation

To rigorously analyze the techniques proposed in the previous sections, we also target the multi-UAV exploration application (see Section 1.1) and conduct both simulations and experiment with real Intel Aero Ready to Fly drones. Thus, we utilize the same setting as in Section 3.6.1 and 3.6.2 and make two adjustments as follows:

- The working environment is represented by a grid: some squares of the grid are selected to be charging stations, and the others are customers. Additionally, UAVs have to follow the grid's edges to move from a place to another (see Figure 4.4 for an example of a 4×4 grid.)
- Although the UAVs are equipped with batteries of $200kJ$, we constrain that each of their flights (since taking off until landing) cannot consume more than $15kJ$.

Recall that we repeated each simulation and experiment 20 times, measured the costs (i.e., energy and time), and considered the mean values as the final costs.

4.6.1 Simulation Results

Analysis of Offline Algorithms

Figures 4.6a, 4.6b, 4.7a, and 4.7b show the results of our simulations of the UAV system. We utilized the MIP solver tool named PolySCIP (Borndörfer *et al.*, 2016) to obtain the optimal solutions for the 3×3 grid (the depot is at $(0, 0)$, the two charging stations are at $(2, 0)$ and $(2, 2)$, and the remaining six nodes are customers) to serve as a reference for what an optimal set of flight paths can achieve. As expected, the

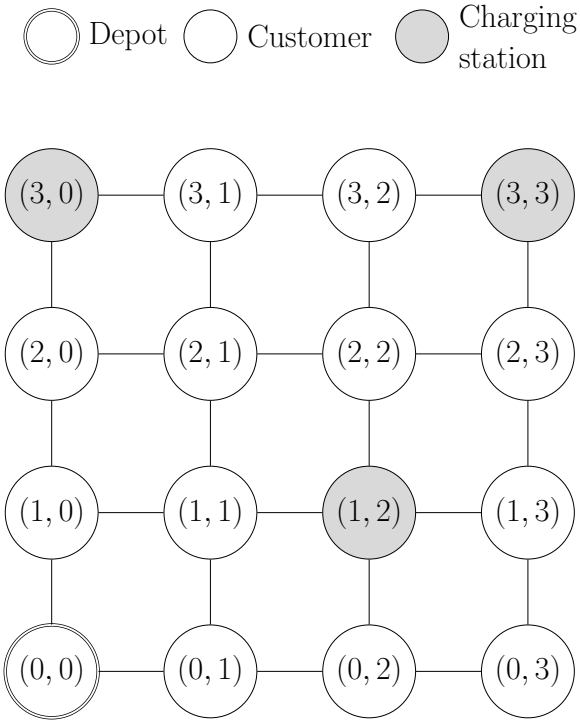
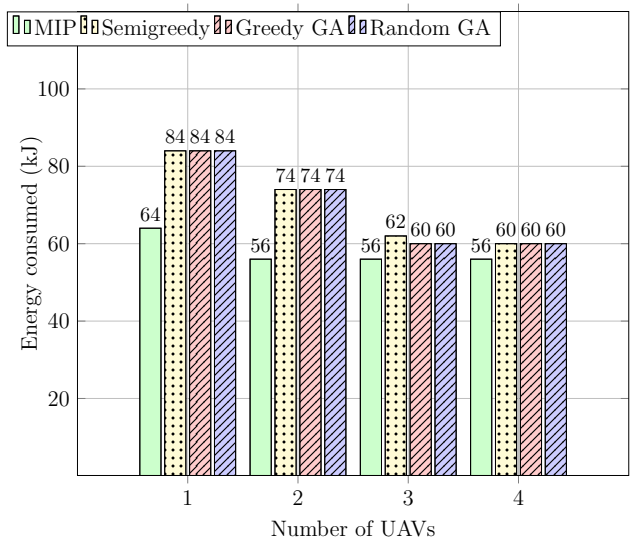
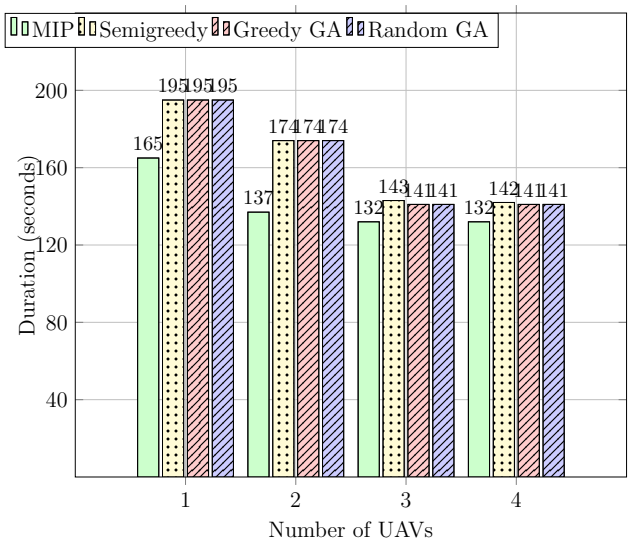


Figure 4.4: Route graph of a 4×4 grid.

plans obtained from MIP cost less energy and time than the genetic algorithms (GAs) and the greedy algorithm (see Figures 4.5a and 4.5b). For larger grids, MIP does not scale well, so we only run the semigreedy algorithm and GAs for the 6×6 (the depot is at $(0, 0)$, and the four charging stations are at $(0, 5), (2, 2), (5, 0)$, and $(5, 5)$) and the 10×10 grids (the depot is at $(0, 0)$, and the nine charging stations are at $(0, 1), (0, 5), (0, 9), (5, 2), (5, 6), (9, 0), (9, 4), (9, 7)$ and $(9, 9)$). Their results are shown in Figures 4.6a, 4.6b, 4.7a, and 4.7b showing that the greedy GA consistently outperforms the random GA (approximately 7%) and the semigreedy algorithm (about 20%) for both time and energy consumption. Our simulations for other grid sizes show the same pattern, but are not presented for reasons of space.

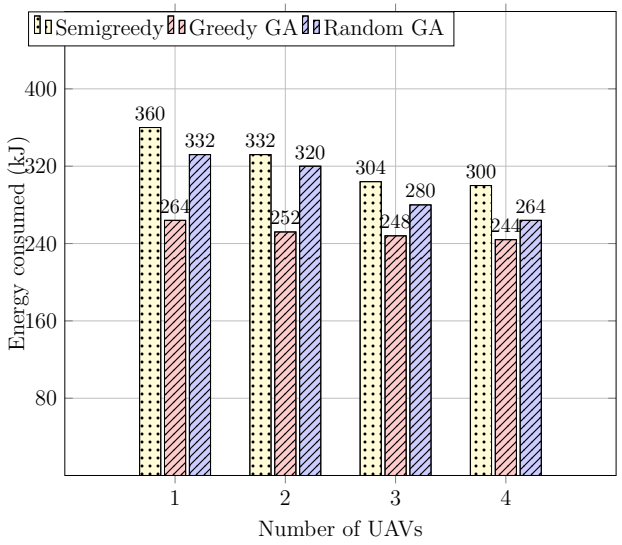


(a) 3×3 Energy Consumption

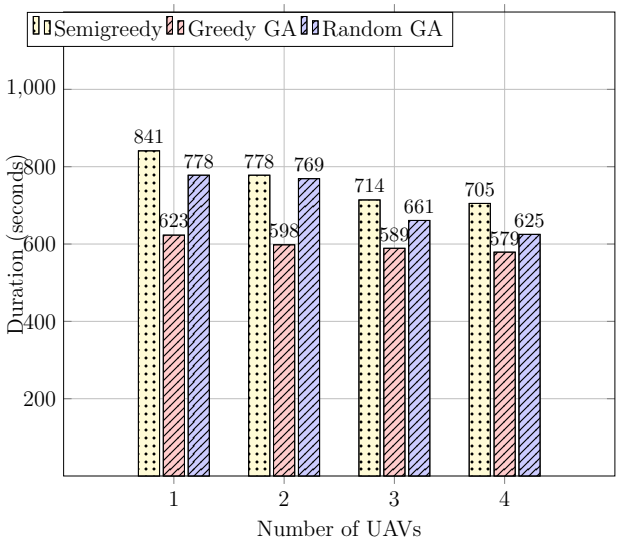


(b) 3×3 Time consumption

Figure 4.5: Simulation results for the OR problem on 3×3 grid.

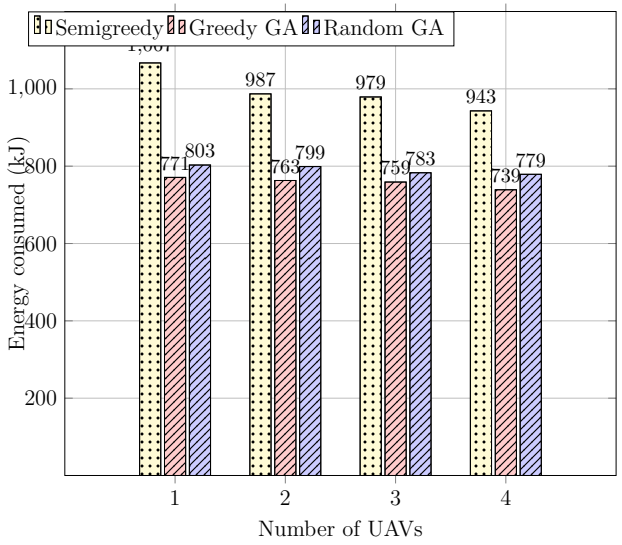


(a) 6×6 Energy Consumption

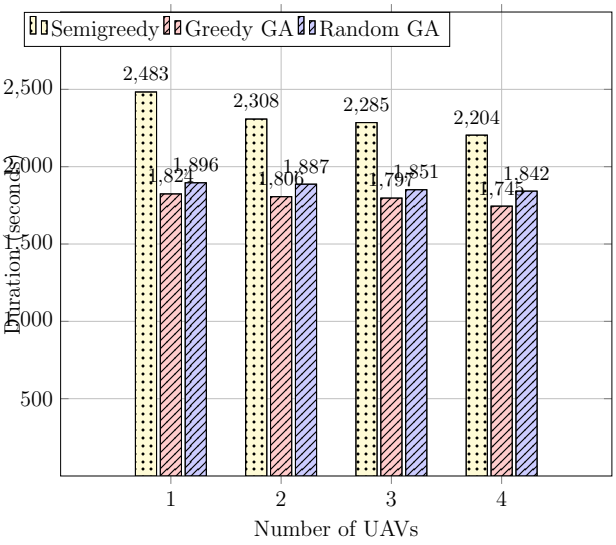


(b) 6×6 Time consumption

Figure 4.6: Simulation results for the OR problem on 6×6 grid.



(a) 10×10 Energy Consumption



(b) 10×10 Time consumption

Figure 4.7: Simulation results for the OR problem on 10×10 grid.

Analysis of the Online Algorithm

In order to analyze our online algorithm, we design the following scenarios. In the first scenario, three UAVs explore a 4×4 grid (see Figure 4.4). If the conditions during the entire operation do not change, then the UAVs complete their mission in $95s$ and spend about total $96kJ$ for the entire fleet. (see Figure 4.8). This solution is obtained by the offline greedy GA. Now, we take the same flight paths and inject a UAV crash at time $19s$. The online algorithm realizes that the current flight paths are not feasible and computes a new plan for the remaining two UAVs. The new flight paths take $114s$ and about total $115kJ$ to complete all the tasks. In the third scenario, we inject an additional UAV crash at time $57s$, and the online algorithm once again computes a new plan which takes $181s$ and about total $190kJ$ to service all the customers.

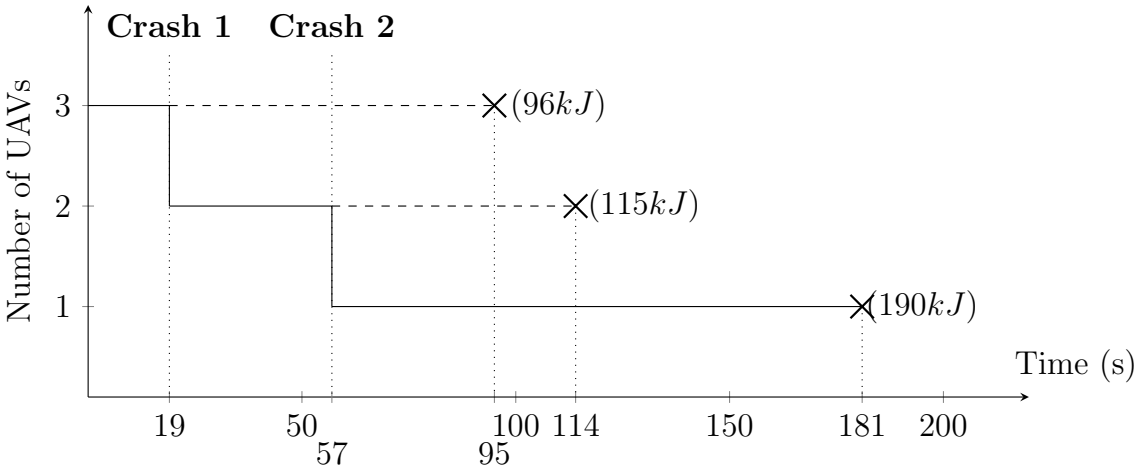
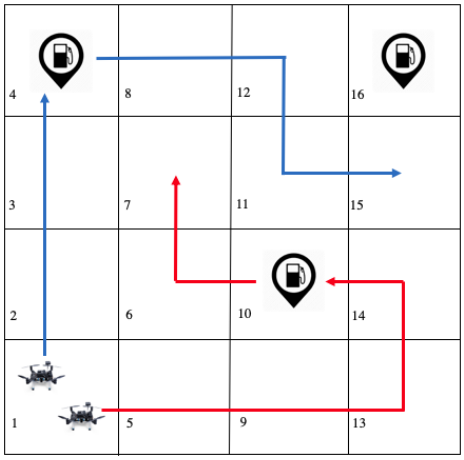


Figure 4.8: Simulation scenarios for the OR online algorithm

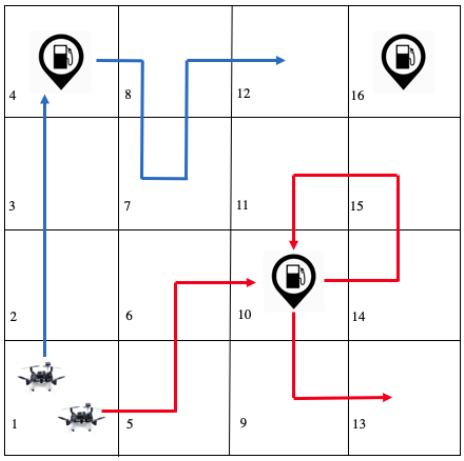
4.6.2 Experiment Results

Recall that in our experiments, (1) we had to deploy the two UAVs flying at two different altitudes due to the safety reasons, (2) when the two UAVs fly close together, the lower UAV spends about 12% more energy to resist the air flow generated by the higher UAVs, and (3) although the UAVs are equipped with batteries of $200kJ$, we constrain that each of their flights (since taking off until landing) cannot consume more than $15kJ$. In the first type of experiments, we disabled the online algorithm. When the UAVs follow the flight paths prescribed by the greedy GA (see Figure 4.9a), the UAV flying in a lower altitude cannot complete its first subwalks, resulting in a pre-mature completion of the mission. The reason for this is that both the UAVs leave from the same depots at the same time, so they are flying very close together (one of top of the other) causing the lower UAV to spend more energy (i.e., to resist the air flow generated by the higher UAVs) than predicted by the greedy GA and reach its energy bound ($15kJ$) too quickly.

Next, we enabled the online algorithm initialized by the same flight paths. The online algorithm effectively conducted appropriate re-planning and serviced the 13 customers (see Figure 4.9b). In another online experiment, we randomly select a UAVs and force it landing to simulate a crash. The crashed UAV then sends a message to the ground station announcing the issue. The online algorithm computed a new plan which reassigns all remaining customers of the crashed drone to the the alive one. Thus, the online algorithm successfully reacts to changes of the physical system as we expected.



(a) Offline solution for the OR problem on 4×4 grid.



(b) Online solution for the OR problem on 4×4 grid.

Figure 4.9: Experiment results of the OR problem.

Chapter 5

Optimal Path Planning in the Presence of Disturbances

In this chapter, we propose the *Optimal Path Planning in the Presence of Disturbances (OPPD)* problem, which aims to design optimal paths for a team of mobile robots in the presence of disturbances. (a brief introduction of the problem was given in Section 1.3) We assume that the robots are homogeneous and have access to a priori known map of the environment, which may contain obstacles. The map is then modeled as an undirected, connected, and finite graph whose nodes are customers, depots, and crossroads (which do not require any service from robots), and edges denote the possibility of traveling. Our ultimate goal is to find an optimal task assignment and path planning for the robots that minimizes energy consumption. To this end, we propose techniques to model disturbance dynamics and execution units' energy consumption, and three *centralized* and one *decentralized* planning algorithms:

- First, we transform our optimization problem into *integer linear programming*

(ILP). This technique produces the optimal result, though it is offline and limited to small problem sizes.

- Next, we introduce two heuristic algorithms based on Nearest Neighbor Heuristic to find sub-optimal solutions. In the first algorithm, assignments for execution units are sequentially constructed, so it will utilize less number execution units if possible. We also propose the second heuristic that aims to construct the missions simultaneously and is suitable for parallelism.

- Finally, we also propose a *decentralized* algorithm based on negotiation.

The rest of this chapter is organized as follows. In Section 5.1, we formally state the problem. Section 5.2 presents our transformation to ILP. Sections 5.3 introduce our two centralized heuristics, while Section 5.4 presents our online decentralized algorithm. Experimental results are analyzed in Section 5.5.

5.1 Problem Statement

In this section, we formally state our OPPD problem. To this end, we start by how to model the physical working environment of robots and identify constraints and objectives of the problem.

5.1.1 Waypoint Graph

In order to model the transitional possibility of robots within a physical environment, we first map the environment to a suitable (2-D or 3-D) square grid. For example, if robots are rovers, a 2-D grid is good enough to model their working environment

(see Figure 5.1a.) In the case of UAVs, we have to use a 3-D grid. Then, we utilize a graph data structure to model transitions. Formally,

Definition 5. *A* waypoint graph *is an undirected, connected, and finite graph* $G = (V, E)$*, where:*

- $V = V_D \cup V_C \cup V_X$ *is the set of all vertices, where* V_D, V_C *and* V_X *are three* disjoint *sets of* depots *(where robots are initially located),* customers *(where a robot have to come to do a task), and crossroads (which do not require any service from robot but may be passed by robots to reach their customers), respectively. This means, each vertex can play only a single role, being either a depot, customer or crossroad.*

- E *is the set of arcs, where an edge* $(v, v') \in E$ *denotes the possibility of traveling between the nodes.* □

The procedure deriving the waypoint graph from a given working area is as follows:

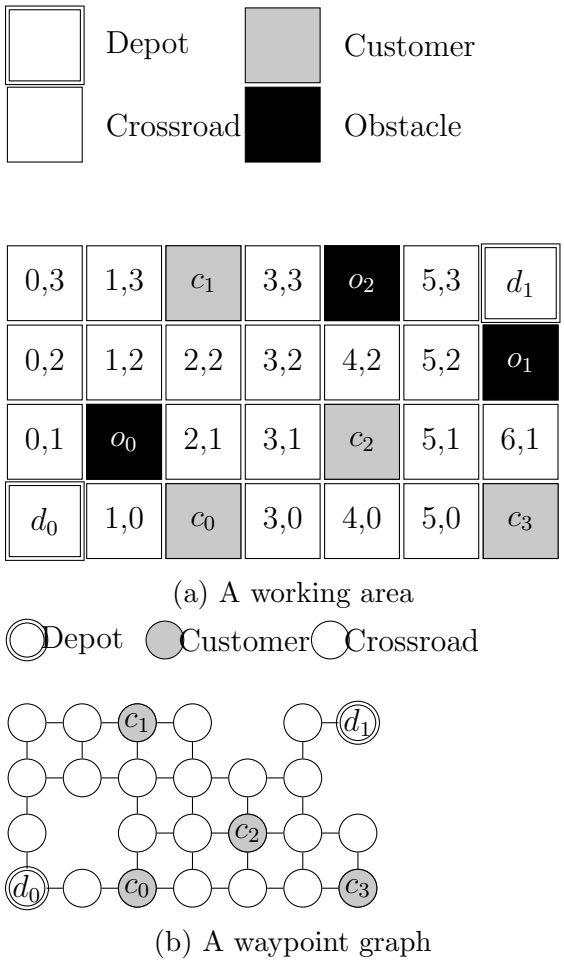
1. Divide the area into a set of adjacent cells (i.e., squares or cubes for 2-D or 3-D grid, respectively),

2. Eliminate cells that cover obstacles,

3. Represent the remaining cells by vertices,

4. Add an arc between a pair of nodes (v, v') if their relative cells are adjacent (i.e., having a common edge or surface in 2-D or 3-D plane, respectively).

5. Remove nodes that are not reachable from any depots (i.e., there is no path from one of the depots to these nodes.)



(a) A working area

(b) A waypoint graph

Figure 5.1: A flight area and its corresponding waypoint graph

For example, Figure 5.1a illustrates a working environment for rovers, where there are two depots (d_0 and d_1), four customers (c_0, c_1, c_2, and c_3), and three obstacles (o_0, o_1, and o_2). The corresponding waypoint graph of the environment is shown in Figure 5.1b, in which the obstacles have been removed.

5.1.2 Disturbances Model

When an execution unit navigates to a customer, there are many unforeseen factors which may expand the required energy of precomputed optimal paths and lead to a need for replanning. In order to model and predict the disturbances dynamics from a software perspective, we propose the use of *Gaussian Process* (GP) and denote it by function $\mathcal{D}$. That is, $\mathcal{D}$ takes as input time t and a node v from a waypoint graph and then predicts the disturbances that happen at v at time t. For example, in the case of wind disturbances, $\mathcal{D}(t, v) = [d_x^v\, d_y^v\, d_z^v]$ will returns a prediction of wind speed in $x-, y-$, and $z-$directions at v.

5.1.3 Energy Model

Deriving a parametric energy model of robots is not a trivial task. For example, although from the actuation perspective, UAVs can be distinguished in only two main categories (i.e., fixed-wing and multi-rotor), there is a wide variety of UAVs, each with different characteristics such as weights, dimensions, propellers, and types of motors. Thus, their energy models are different. Though there is an extensive line of research constructing energy model for different UAVs, and some have already taken into account the wind disturbances (e.g., (Bezzo *et al.*, 2016; Morbidi *et al.*, 2016; Zeng and Zhang, 2017)), most of the work is from a control-theoretic and avionics perspectives. Only (Di Franco and Buttazzo, 2016), to our knowledge, proposes an approach to derive the energy model of an IRIS quad-copter from a software perspective by conducting a series of experiments on the drone. The model is expressed as a function of the drone's speed under different simple flight conditions such as horizontal flight, climbing, descending, and hovering but does not include any external

disturbances (e.g., wind). In this research, we extend this approach with external disturbance.

For generalization, we abstract the energy model of the execution units by two functions:

- **Traveling estimator $\mathcal{T}$.** This function takes as input a time t, an edge $e = (v, v') \in E$, and a disturbance model $\mathcal{D}$ and predicts an amount of energy and time that an execution unit needs to move from v to v'. Formally,

$$(\varepsilon, \tau) = \mathcal{T}(t, e, \mathcal{D})$$

  where ε and τ are the amount of energy and time possibly consumed by an execution unit to travel from v to v', respectively. We also use $\mathcal{T}(t, e, \mathcal{D}).\varepsilon$ (respectively, $\mathcal{T}(t, e, \mathcal{D}).\tau$) to indicate traveling energy (respectively, time.)

- **Execution estimator $\mathcal{S}$.** This function takes as input time t, a customer $c \in V_C$, and a disturbances model $\mathcal{D}$ and then returns an estimation of energy and time that an execution unit needs to service a customer c. Formally,

$$(\varepsilon, \tau) = \mathcal{S}(t, c, \mathcal{D})$$

  where ε and τ are the amount of energy and time possibly consumed by an execution unit to service c, respectively. We also use $\mathcal{S}(t, c, \mathcal{D}).\varepsilon$ (respectively, $\mathcal{S}(t, c, \mathcal{D}).\tau$) to indicate traveling energy (respectively, time.)

Now, we are interested in finding a planning function for minimizing the gloabl energy consumption of execution units. First, let $m^u = (v_0^u, t_0)(v_1, t_1) \cdots (v_n, t_n)$ be a

mission assigned to execution unit $u \in \mathbb{U}$, where $v_i \in V$, t_i is the time that u arrives at v_i for $i \in [0, n]$, and $(v_i; v_{i+1}) \in E$ for all $i \in [0, n)$ (i.e., a unit must go through adjacent nodes in G.) We use $m^u(i).v$, $m^u(i).t$, and $|m^u|$ to denote the v_i, t_i and the length of m^u, respectively. Note that a node $v \in V$ can appear multiple times in m^u. Recall that v_0^u is the node where execution unit u is initially located (the depot of u.) We denote the set of all possible missions by M. Then, we aim to design a *plan function* $\mathcal{F} : \mathbb{U} \to M$, that maps an execution unit to a mission and is subject to the following constraints:

5.1.4 Constraints

First, we denote the set of customers serviced by execution unit u taking m^u by $C(m^u)$. Formally,

$$C(m^u) = \big\{ m^u(i) \mid i \in [0, n] \wedge m^u(i).v \in V_C \big\}.$$

We now detail the constraints or our proposed problem.

Affinity. Every execution unit is only allowed to service specific customers as determined by its affinity $\mathcal{A}_u$:

$$\forall u \in \mathbb{U} : C(\mathcal{F}(u)) \subseteq \mathcal{A}_u \tag{5.1.1}$$

Completion. All customers must be serviced exactly once.

$$\begin{gathered} \bigcup_{u \in \mathbb{U}} C(\mathcal{F}(u)) = V_C \\ C(\mathcal{F}(u)) \cap C(\mathcal{F}(u')) = \emptyset \quad \text{for each distinct } u, u' \in \mathbb{U} \end{gathered} \tag{5.1.2}$$

Energy bound. Each execution unit should respect its energy bound. First, let $\mathbb{C}(m^u)$ denote the total energy consumption of a mission m assigned to an execution unit u:

$$\mathbb{C}(m^u) = \sum_{c \in C(m^u)} \mathcal{S}\big(t_c, c, \mathcal{D}\big).\varepsilon + \sum_{i=0}^{n-1} \mathcal{T}\big(m^u(i).\tau, (m^u(i).v, m^u(i+1).v), \mathcal{D}\big).\varepsilon \tag{5.1.3}$$

where t_c is the time when u arrives c determined by m^u. Thus, the following constraint enforces that an execution unit obeys its energy bound:

$$\mathbb{C}(\mathcal{F}(u)) \leq \mathcal{B}_u \quad \text{for } u \in \mathbb{U} \tag{5.1.4}$$

5.1.5 Optimization Objective

Our optimization objective is to to minimize the global energy consumption. That is:

$$min\left[\sum_{u \in \mathbb{U}} \mathbb{C}\big(\mathcal{F}(u)\big)\right]$$

5.2 Integer Linear Program Transformation

In this section, we transform our problem into integer linear programming (ILP). We start by identifying the challenges in the transformation:

- *Incomplete waypoint graphs.* This causes that an execution unit may have to travel through some crossroad nodes multiple times to reach its customers, which creates cycles in its walks and makes the transformation harder.
- *Effects of disturbances.* This cause that (1) traveling cost of an edge varies from different directions, and (2) both traveling and service costs vary from time to time.

From the above challenges and the problem's constraints (see Section 5.1.4), it is clear that in the optimal solution only crossroad nodes may be visited more than once. Thus, we propose to utilize a graph data structure to abstract away crossroad nodes (to eliminates cycles in the optimal solution). Additionally, we consider discrete time with sampling time t_s, divide the operational time into time intervals $T = \{t_0 = 0, t_1, \cdots, t_m\}$. In this way, the service and traveling estimators can be viewed as known step functions of the intervals (i.e., the service and traveling costs are known constants.)

Now, the problem can be formulated on a weighted directed multigraph, called *route graph* whose nodes are the depots and customers from the corresponding waypoint graph, and edges (connecting these nodes) encode shortest paths. By shortest paths, we mean paths that cost less traveling energy. Due to the presence of obstacles and especially disturbances, these paths are likely not straight lines between the nodes. In order to obtain the weight (i.e., the traveling energy and time) of an edge

(v, v'), we can explore an adaption of existing shortest path algorithms (e.g., Dijkstra (Dijkstra, 1959), $A^\star$ (Hart *et al.*, 1968), RRT (Kuffner Jr and LaValle, 2000)) on the corresponding waypoint graph. We denote this algorithm by function Shortest taking as input an interval $t \in T$ and a pair of distinct nodes v and v', and then returning an amount of energy and time that an execution unit needs to move from v to v'. Formally,

$$(\varepsilon, \tau) = \text{Shortest}(t, v, v')$$

where ε and τ are the amount of energy and time possibly consumed by an execution unit to travel from v to v', respectively. We also use $\text{Shortest}(t, v, v').\varepsilon$ (respectively, $\text{Shortest}(t, v, v').\tau$) to indicate traveling energy (respectively, time.) The formal definition of a route graph is as follows:

Definition 6. *Let $G = (V, E)$ be the waypoint graph as defined in Definition 5. The corresponding* route graph *is a finite complete weighted directed multigraph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where:*

- *The set of all vertices $\mathcal{V}$ is $V_C \cup V_D$. Recall that V_C and V_D are the set of customers and depots in G, respectively.*
- *The set of all directed edges is $\forall v, v' \in \mathcal{V}, t \in T : (v, v', t) \in \mathcal{E}$. Recall that T is the set of all intervals, and the interval t denotes the time when execution unit departs from v to v'.*
- *Finally, the weight w_e of edge $e = (v, v', t) \in \mathcal{E}$ is a pair (ε, τ), where $v, v' \in \mathcal{V}, t \in T$, $t' = \text{Shortest}(v, v', t).\tau$, $\varepsilon = \text{Shortest}(v, v', t).\varepsilon + \mathcal{S}(t', v').\varepsilon$, $\tau = t' + \mathcal{S}(t', v').\tau$. That is, the weight includes both the traveling and service costs.*

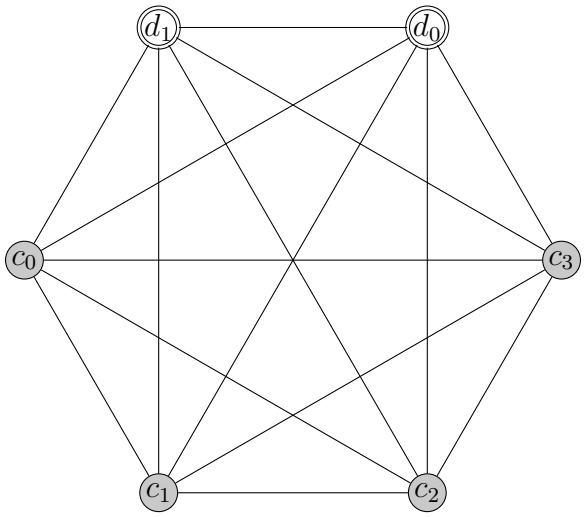


Figure 5.2: A route graph

□

For example, Figure 5.2 illustrates a route graph of the waypoint graph in Figure 5.1b. For simplicity, we consider only a single interval t_0 and utilize an undirected edge (v, v', t_0) to represent two arcs (v, v', t_0) and (v', v, t_0).

Now, we detail our ILP. Let $\mathbb{U}$ be a finite set of execution units, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a given route graph, and $T = \{t_0 = 0, t_1, \cdots, t_m\}$ be the set of all intervals. First, the following are decision variables in our ILP instance:

5.2.1 Decision Variables

- $x^u_{(v,v',t)}$: Boolean variable indicating if execution unit u travels from v to v', for each $(v, v', t) \in \mathcal{E}$.

- $t^u_{(v,t)}$: Non-negative integer variable indicating the time when execution unit u arrives v, where $v \in \mathcal{V}$ and $t \in T$.

Second, the constrains are as follows:

5.2.2 Constraints

Affinity. An execution unit can only do tasks in its affinity set:

$$\sum_{v \notin \mathcal{A}_u} x^u_{(v',v,t)} = 0 \quad \text{for all } u \in \mathbb{U}, v \in \mathcal{V}, t \in T \tag{5.2.1}$$

Recall that T is the set of all intervals.

Mission validity. An execution unit u

- must leave from its depot (v^u_0) at t_0:

$$\sum_{(v^u_0,v,t) \in \mathcal{E}} x^u_{(v^u_0,v,t)} \leq 1 \tag{5.2.2}$$

$$\sum_{(v^u_0,v,t) \in \mathcal{E}} x^u_{(v^u_0,v,t)} = 0 \qquad \forall t \in T \setminus \{t_0\} \tag{5.2.3}$$

- continuously goes through adjacent nodes:

$$0 \leq \sum_{(v,v',t) \in \mathcal{E}} x^u_{(v,v',t)} - \sum_{(v',v'',t') \in \mathcal{E}} x^u_{(v',v'',t')} \leq 1 \qquad \forall v' \in V \setminus \{v^u_0\} \tag{5.2.4}$$

$$t^u_{(v,t)} - t^u_{(v',t')} + t_m \cdot (x^u_{(v,v',t)} - 1) \leq -w_{(v,v',t)}.\tau \quad \forall v, v' \in V_C, \forall t, t' \in T \tag{5.2.5}$$

$$t^u_{(v,t)} - t^u_{(v',t')} - t_m \cdot (x^u_{(v,v',t)} - 1) \geq -w_{(v,v',t)}.\tau \quad \forall v, v' \in V_C, \forall t, t' \in T \tag{5.2.6}$$

$$t^u_{(v,t)} \cdot x^u_{(v,v',t^u_{(v,t)})} - t^u_{(v,t)} \geq 0 \qquad \forall v, v' \in V_C \tag{5.2.7}$$

$$t^u_{(v,t)} \in T \qquad \forall v \in V_C \tag{5.2.8}$$

Recall that t_m is the maximum interval (i.e., a constant), $w_{(v',v,t)}$ is the weight of edge $e = (v', v, t)$. Equation (5.2.4) is the *flow constraint* enforcing that execution unit u can only leave a node only after having arrived it. Equation (5.2.5)

and (5.2.6) are called *subtour elimination constraints* which are adapted from the ones proposed by Miller-Tucker-Zemlin in (Miller *et al.*, 1960) and is utilized to eliminate cycles (i.e., subtours). When they are combined with Equation 5.2.7, they ensure that only one of the outgoing arcs of v is selected, and the arrival time $t^u_{(v',t)}$ at the new node v' is equal to the traveling time. The Equation 5.2.7 contains a product of two decision variables ($t^u_{(v,t)}$ and $x^u_{(v,v',t^u_{(v,t)})}$) making our transform nonlinear. In order to linearize it, we replace the product with a new variables $y^u_{(v,v',t^u_{(v,t)})}$ and add the following constraints:

$$y^u_{(v,v',t^u_{(v,t)}))} \leq t_m \cdot x^u_{(v,v',t^u_{(v,t)})} \tag{5.2.9}$$

$$y^u_{(v,v',t^u_{(v,t)}))} \leq t^u_{(v,t)} \tag{5.2.10}$$

$$y^u_{(v,v',t^u_{(v,t)}))} \geq t^u_{(v,t)} - t_m \cdot (1 - x^u_{(v,v',t^u_{(v,t)})}) \tag{5.2.11}$$

- does not return to the depot:

$$\sum_{(v,v^u_0,t)\in\mathcal{E}} x^u_{(v,v^u_0,t)} = 0 \quad \forall v \in \mathcal{V},\, t \in T \tag{5.2.12}$$

Completion. All customers must be visited exactly once:

$$\sum_{u\in\mathbb{U}} x^u_{(v',v,t)} = 1 \qquad\qquad \forall v \in V_C \tag{5.2.13}$$

Energy bound. Every execution unit must operate within its energy bound. That

is,

$$\sum_{(v',v,t)\in\mathcal{E}} w_{(v',v,t)}.\varepsilon \times x^u_{(v',v,t)} \leq \mathcal{B}_u \quad \forall u \in \mathbb{U} \tag{5.2.14}$$

5.2.3 Objective Function

As mentioned in Section 5.1, our problem requires optimizing energy consumption. Formally,

$$\min \sum_{u\in\mathbb{U}} \sum_{(v',v,t)\in\mathcal{E}} w_{(v',v,t)}.\varepsilon \times x^u_{(v',v,t)} \tag{5.2.15}$$

5.3 Heuristic Algorithm

In this section, we introduce our greedy algorithms based on the Nearest Neighbor Heuristic (NNH) (Bellmore and Nemhauser, 1968). The NNH utilizes a loop to sequentially select a customer which has not been assigned to any execution units and has the lowest insertion cost to assign to an execution unit. Recall that the insertion cost of a customer includes both the traveling and servicing cost required to append the customer to the end of the partially build mission of an execution unit.

5.3.1 Sequential Mission Construction Heuristic

In this heuristic (Algorithm 8), the mission of each execution unit is sequentially constructed, and a new execution unit is introduced only when the remaining unvisited customers cannot be assigned to the current execution unit. Therefore, some execution unit may not be utilized if the total energy bound of execution units greatly exceeds the needed energy. The algorithm takes as input a waypoint graph $\mathcal{G}$ and

Algorithm 8: Sequential Mission Construction for OPPD

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $\mathbb{U}$
Output: **True** or **False**

```
1  m^u ← (v_0^u, 0) ∀u ∈ 𝕌
2  T̄ ← V_C
3  foreach u ∈ 𝕌 do
4      finished ← False
5      while finished = False do
6          (v, t) ← the last element in m^u
7          C^u[v'] ← w_(v,v',t).ε ∀v' ∈ T̄
8          v' ← argmin(C^u)
9          if remaining energy of u ≥ w_(v,v',t).ε then
10             t' ← w_(v,v',t).τ
11             insert (v', t') into the end of m^u
12             remove v' from T̄
13         else
14             finished ← True
15 if T̄ ≠ ∅ then
16     return ∅
17 return m^u ∀u ∈ 𝕌
```

a set of execution units $\mathbb{U}$, and then returns missions for the execution units. The algorithm works as follows:

1. Initialize mission for each unit (i.e., each mission is started with the depot of the execution unit at time 0.) (Line 1)

2. Initialize the set of unserviced customers $\bar{T}$ (Line 2).

3. Utilize a loop to sequentially assign customers to each unit (Lines 5 -14). At each iteration,

   (a) Choose the cheapest customer to service (Line 8)

(b) If the current unit has enough energy to service the customer, append the customer to the end of the unit's mission (Line 9)

(c) Otherwise, terminate the loop and switch to the next execution unit (Line 14).

4. Return the missions of execution units.

The heuristic may indicate insufficient energy (returning an empty set of missions) when in fact a feasible solution does exist due to its myopia. However, it will always find a feasible solution if the energy bound of execution units is much larger than the total requirement of the customers.

5.3.2 Simultaneous Mission Construction Heuristic

In this heuristic (Algorithm 9), the missions of execution units are simultaneously constructed, and a customer is assigned to an execution unit which has the lowest insertion cost. This also means, all available execution units are likely utilized even if the total energy bound of execution units greatly exceeds the needed energy. The algorithm takes as input a waypoint graph $\mathcal{G}$ and a set of execution units $\mathbb{U}$, and then returns missions for the execution units. The algorithm works as follows:

1. Initialize mission for each unit (i.e., each mission is started with the depot of the execution unit at time 0) (Line 1).

2. Initialize the set of unserviced customers $\bar{T}$ (Line 2).

3. Initialize the set of available execution units $\bar{\mathbb{U}}$ (Line 3).

4. Utilize a loop to sequentially assign unvisited customers (Lines 4 -20). At each iteration,

Algorithm 9: Simultaneous Mission Construction for OPPD

Input: $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, $\mathbb{U}$
Output: **True** or **False**

$m^u \leftarrow (v_0^u, 0)\ \forall u \in \mathbb{U}$
$\bar{T} \leftarrow V_C$
$\bar{\mathbb{U}} \leftarrow \mathbb{U}$
while $\bar{\mathbb{U}} \neq \emptyset \wedge \bar{T} \neq \emptyset$ **do**
    $C \leftarrow \emptyset$
    foreach $u \in \bar{\mathbb{U}}$ **do**
        $(v, t) \leftarrow$ the last element in m^u
        $C[(u, v')] \leftarrow w_{(v,v',t)}.\varepsilon\ \forall v' \in \bar{T}$
    finished $\leftarrow$ **False**
    while *finished* $=$ **False** $\wedge\ \bar{\mathbb{U}} \neq \emptyset$ **do**
        $(u, v') \leftarrow \text{argmin}(C)$
        $(v, t) \leftarrow$ the last element in m^u
        if *remaining energy of* $u \geq w_{(v,v',t)}.\varepsilon$ **then**
            $t' \leftarrow w_{(v,v',t)}.\tau$
            insert (v', t') into the end of m^u
            remove v' from $\bar{T}$
            finished $\leftarrow$ **True**
        else
            remove u from $\bar{\mathbb{U}}$
            remove (u, v'') from C, $\forall v'' \in \bar{T}$
if $\bar{T} \neq \emptyset$ **then**
    return $\emptyset$
return $m^u\ \forall u \in \mathbb{U}$

(a) Initialize a set C to store all the insertion costs (Line 5).

(b) Choose the cheapest pair of execution and customer that has the lowest insertion cost (Line 11).

(c) If the selection yields a feasible solution (i.e., not violating energy bound), then commit the selection (Line 17).

(d) Otherwise, repeat Step 4b until there is no available execution unit.

5. Return the missions of execution units.

The heuristic also may indicate insufficient energy (returning an empty set of missions) when in fact a feasible solution does exist due to its myopia. Although both aforementioned heuristics can find a solution in $O(|\mathbb{U}| \cdot |\mathcal{V}|)$ time in the worst case, Algorithm 9 is more parallelizable. Recall that $\mathcal{V}$ is the set of vertices in the given route graph $\mathcal{G}$, and $\mathbb{U}$ is the set of all execution units.

5.4 Online Algorithm

In this section, we propose an online and decentralized algorithm. To this end, we first assume that (1) each execution unit is equipped with a WiFi module allowing to communicate with one another over a reliable network (i.e., no messages are altered or spuriously introduced), (2) each execution unit has the unique integer ID which can be utilized to determine the priority between execution units (i.e., the one with lower ID has higher priority.), and (3) some execution units may leave the team during execution time (because of crashes).

Next, we discuss the input of the proposed algorithm. First, when one of the execution units notices environmental changes that do not conform with its disturbances

models at time t, there could be some (or all) customers which have been serviced and must be eliminated before invoking the algorithm. We denote the route graph modeling the problem at time t by $\tilde{G}$ (note that the weights of its edges are also updated wrt. the currently observed disturbances). Final, let us consider a mission $m^u = (v_0^u, t_0)(v_1, t_1) \cdots (v_n, t_n)$ assigned to execution unit u. If u locates at v_i at the time t (if it is traveling to v_i, we still consider that u is at v_i), where $i \in [0, n]$, then the *partially executed mission* of u is $\tilde{m}^u = (v_i^u, t_i) \cdots (v_n, t_n)$.

Now, we introduce Algorithm 10 which is executed by each unit $u \in \bar{\mathbb{U}}$ and takes as input a graph $\tilde{G}$, and its partially executed mission $\tilde{m}^u$, and then produces its missions after negotiating with other functional execution unit. The algorithm works a follows:

1. Simulate the current mission to check if the execution unit has enough energy to finish its mission. If it is still possible, $\bar{T}_u$ is an empty set. Otherwise, $\bar{T}_u$ contains customers that are in $\tilde{m}^u$ and cannot be visited by u (Line 1).

2. Update the current mission of the execution unit w.r.t to $\bar{T}_u$. That is, we remove every customer $c \in \bar{T}_u$ from $\tilde{m}^u$ (Line 2).

3. Broadcast a message to detect the set of functional execution units $\bar{\mathbb{U}}$ (Line 3).

4. Communicate with functional execution units to get all unassigned customers $\bar{T}$ (Line 4).

5. Utilize a loop to sequentially assign the customers to the units (Lines 5 -23). At each iteration,

   (a) Choose the cheapest customer as a candidate (Line 8)

Algorithm 10: OPPD - Online decentralized algorithm for each unit u

Input: $\tilde{G}, \tilde{m}^u$
Output: $\tilde{m}^u$

$\bar{T}_u \leftarrow \text{Simulate}(\tilde{m}^u)$
update $\tilde{m}^u$
$\tilde{\mathbb{U}} \leftarrow$ the set of functional execution units
$\bar{T} \leftarrow \bigcup_{u' \in \tilde{\mathbb{U}}} \bar{T}_{u'}$
while $\bar{T} \neq \emptyset$ **do**
    $(v, t) \leftarrow$ last element in $\tilde{m}^u$
    $C^u[v'] \leftarrow w_{(v,v',t)}.\varepsilon \; \forall v' \in \bar{T}$
    $v' \leftarrow \text{argmin}(C^u)$
    if *remaining energy of* $u \geq C^u[v']$ **then**
        send $(u, v', C^u[v'])$
        $C_{v'} \leftarrow \bigcup_{u' \in \tilde{\mathbb{U}} \setminus \{u\}} \{C^{u'}[v']\}$
        if $C^u[v'] < \min(C_{v'})$ **then**
            $t' \leftarrow w_{(v,v',t)}.\tau$
            insert (v', t') into the end of $\tilde{m}^u$
        else if $C^u[v'] = \min(C_{v'})$ **then**
            $U_{v'} \leftarrow$ the set of units that has the same insertion cost of v'
            if *u has the highest priority among $U_{v'}$* **then**
                $t' \leftarrow w_{(v,v',t)}.\tau$
                insert (v', t') into the end of $\tilde{m}^u$
        remove v' from $\bar{T}$
    else
        broadcast (u, NIL)
        break
return $\tilde{m}^u$

(b) If the current unit has enough energy to service the customer, communicate with other available to determine which execution unit will service the selected customer (Line 10). The selected unit is the one with the lowest insertion cost (Line 12), and execution units' priorities will be utilized to break any ties if necessary (Line 17).

(c) Otherwise, send a message informing that u has used up its energy and will not accept any other mission (Line 22).

6. Return the missions of execution units.

5.5 Evaluation

To rigorously analyze techniques proposed in the previous sections, we, once again, utilize simulations and experiments of the multi-UAV exploration application (see Section 1.1). We also employ the same setting as in Section 3.6.1 and 3.6.2 and make three adjustments as follows:

- The working environment is is represented by a 3-D square grid (see Figure 5.3 for an example of a 3×2 grid.) Additionally, the grid's edges limit the maneuver of a UAV to six directions: up, down, east, west, north, and south.
- The experiment platform includes a $20''$ box fan capable of generating wind speed up to $3m/s$ and two cardboard boxes as stationary obstacles.
- Although the UAVs can fly up to $20m/s$, we limit their speed down to $5m/s$, which greatly increases the impact of the wind (generated by the box fan) on qualification of the pre-computed optimal paths of UAVs.

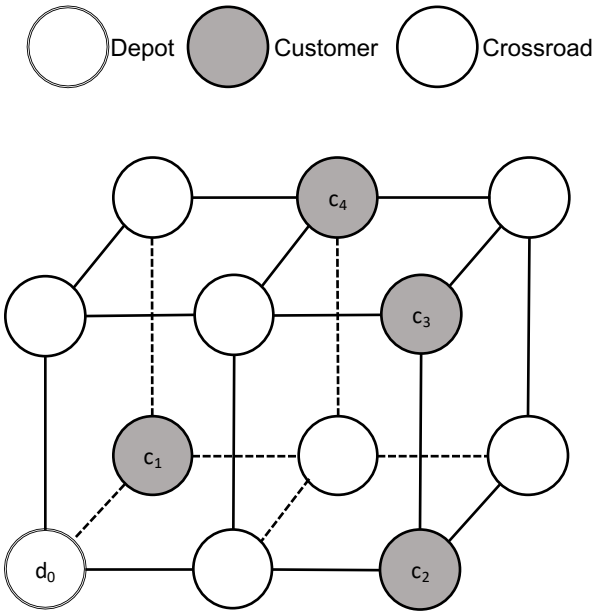


Figure 5.3: Waypoint graph of a 3×2 grid.

Recall that we repeated each simulation and experiment 20 times, measured the costs (i.e., energy and time), and considered the mean values as the final costs.

5.5.1 Disturbances Model Construction

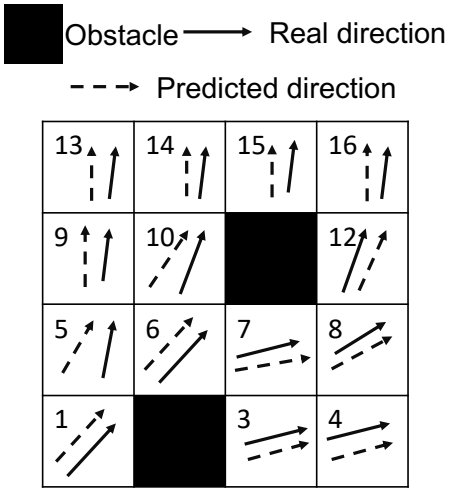
In our experiment, we considered wind as the only external disturbances and was created through a $20''$ box fan capable of generating wind speed up to $3m/s$. To construct wind models, we carried a set of experiments:

- In the first experiment, we aimed to derive the wind dynamics as a function of location. To this end, we had the fan generate wind at constant speed and utilize a wind speed meter (see Figure 5.4) to measure the $x-$ and $y-$direction wind speed at each square of the working area. Then, we developed a Python application utilizing Scikit-learn library (Pedregosa *et al.*, 2011) to construct a Gaussian Process modeling the wind. Figure 5.5a shows the real wind direction

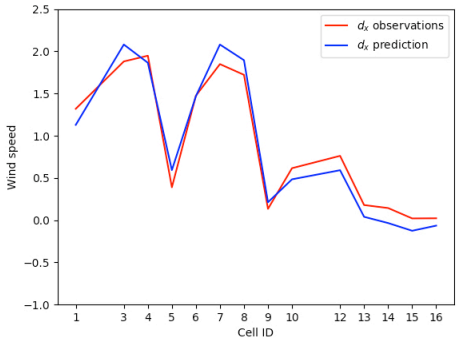
and one predicted by our model while the detail speeds in $x-$ and $y-$direction at each cell are shown in Figures 5.5b and 5.5c, respectively. For simplicity, we utilize a 2-D grid instead of 3-D one. Note that this model is mainly used for our simulation to compare the performance of our heuristic algorithms and the ILP.

Figure 5.4: A wind speed meter.

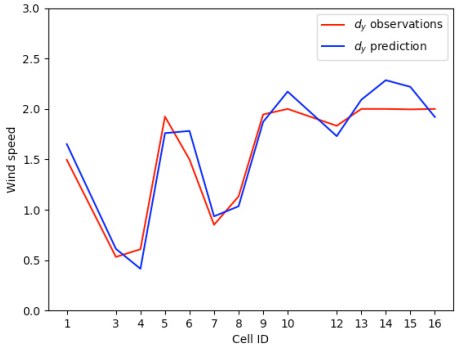
- In the second experiment, we manage derive the the wind dynamics as a function of time and location. That is, we had the fan generate wind at different speed at different time interval. This model is mainly used for our empirical experiment to evaluate how well our online algorithm responses to the real changes of the environment.



(a) Wind direction.



(b) Wind speed in $x-$direction.



(c) Wind speed in $y-$direction.

Figure 5.5: Wind measure and prediction.

5.5.2 Energy Model Construction

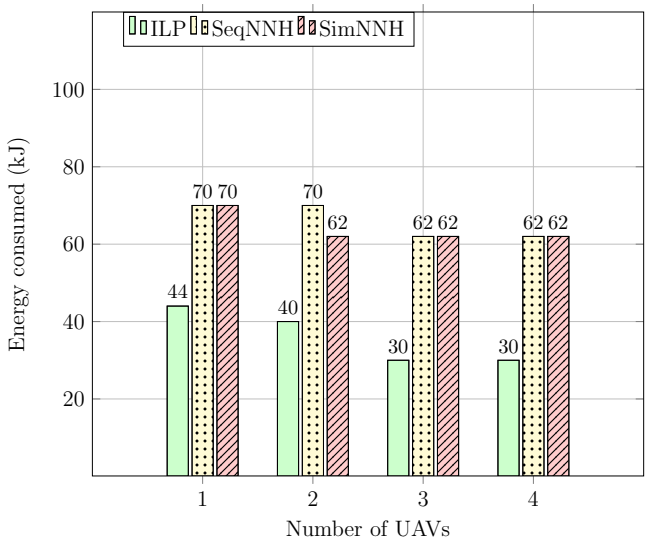
In this research, we employed Intel Aero drones powered with a 3S LiPo battery ($11.1V5Ah$), and weighted approximately $1.5Kg$ in total. In order to derive the energy model of the drone, we carried out a set of experiments:

- In the first experiment, we aim to derive the power consumption as a service function (i.e., $\mathcal{S}$). Thus, we programmed the drone to fly in different conditions such as taking off, landing and hovering with and without the presence of disturbances.

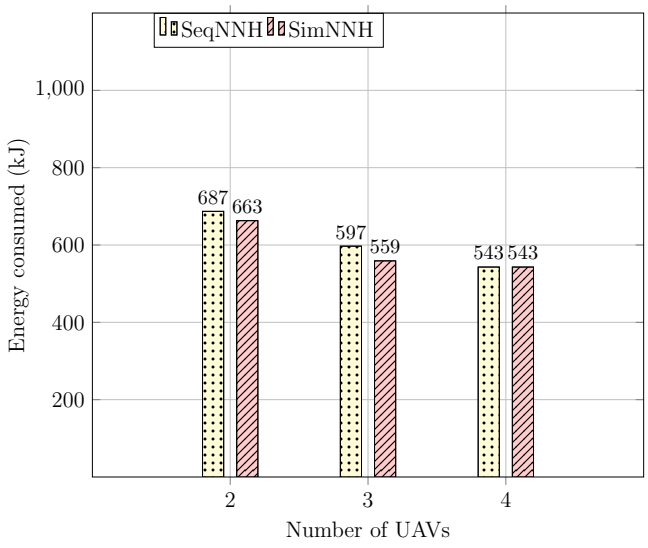
- In the second experiment, we manage to derive the power consumption as a traveling function. Therefore, we programmed the drone to fly at the maximum speed from one place to another with and without the presence of disturbances. The consumed power was then derived for each type of transition (e.g., flying vertically up and down, horizontally to different distances.)

5.5.3 Simulation Results

Figure 5.6 shows the results of our simulations of the proposed offline algorithms. We utilized the CPLEX solver to obtain the optimal solutions for the $3 \times 3 \times 2$ grid (the depots is at $(0, 0, 0)$ and $(2, 2, 0)$, three customers are at $(2, 0, 0), (1, 1, , 1)$, and $(0, 1, 1)$, and the remaining seven nodes are crossroads) to serve as a reference for what an optimal set of flight paths can achieve. As expected, the plans obtained from ILP cost less energy and time than the heuristic algorithms (see Figures 5.6a). For larger grids, ILP does not scale well, so we only run the heuristics algorithm for the 10×10 grid (there are two depots, and 20 customers. Their results are shown

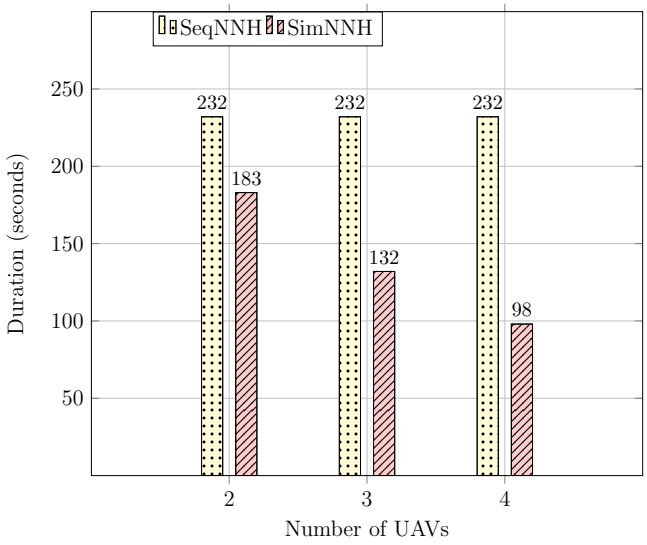


(a) 3×3 Energy Consumption



(b) 10×10 Energy Consumption

Figure 5.6: Simulation results for the OPPD problem on 3×3, and 10×10 grids.



(c) 10×10 Computational Time

Figure 5.6: Simulation results for the OPPD problem on 3×3, and 10×10 grids (cont.).

in Figures 5.6b. Because the both algorithms are based on the same heuristic, the energy consumption of their plans approximately equal. However, Figure 5.6c shows that Algorithm 9 is consistently faster than Algorithm 8 due to its parallelism.

Next, we design the following scenarios to evaluate our online algorithm. In the first scenario, three UAVs explore a $4 \times 4 \times 2$ grid (the depot is at $(3, 3, 0)$, eight customers are at $(2, 0, 0), (1, 1, 1)$, $(1, 1, 0)$, $(3, 0, 1)$, $(2, 2, 1)$, $(0, 3, 1)$, $(0, 2, 0)$, and $(3, 1, 0)$, two obstacles are at $(1, 0, 0)$ and $(2, 2, 0)$, and the remaining nodes are cross-roads). If the conditions during the entire operation do not change, then the UAVs complete their mission in $92s$ and spend about total $107kJ$ for the entire fleet. (see Figure 5.7). This solution is obtained by Algorithm 9. Now, we take the same flight paths and inject a UAV crash at time $25s$. The online algorithm realizes that the current flight paths are not feasible and computes a new plan for the remaining two UAVs. The new flight paths take $141s$ and about total $132kJ$ to complete all the

tasks. In the third scenario, we inject an additional UAV crash at time $67s$, and the online algorithm once again computes a new plan which takes $191s$ and about total $200kJ$ to service all the customers.

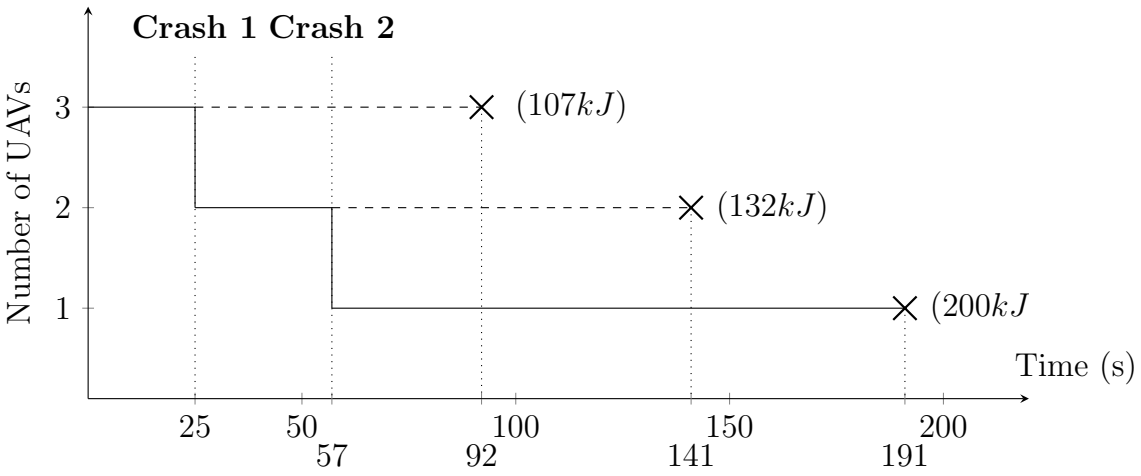


Figure 5.7: Simulation scenarios for the OPPD problem online algorithm

5.5.4 Experiment Results

In the experiments, the lab is then modeled by a 3-D square gird of $4 \times 4 \times 2$. The grid consists of a single depot at $(3, 3, 0)$, eight customers at $(2, 0, 0), (1, 1, 1)$, $(1, 1, 0)$, $(3, 0, 1)$, $(2, 2, 1)$, $(0, 3, 1)$, $(0, 2, 0)$, and $(3, 1, 0)$, and two obstacles at $(1, 0, 0)$ and $(2, 2, 0)$ (the remaining nodes are crossroads) (see Figure 5.8a). We employed the fan at the bottom left corner of the room to generate wind, and two Intel Aero drones to do some tasks at the customers. The ultimate goal of our experiments is to demonstrate the effectiveness of our offline and online algorithms. To this end, we carried two types of experiments:

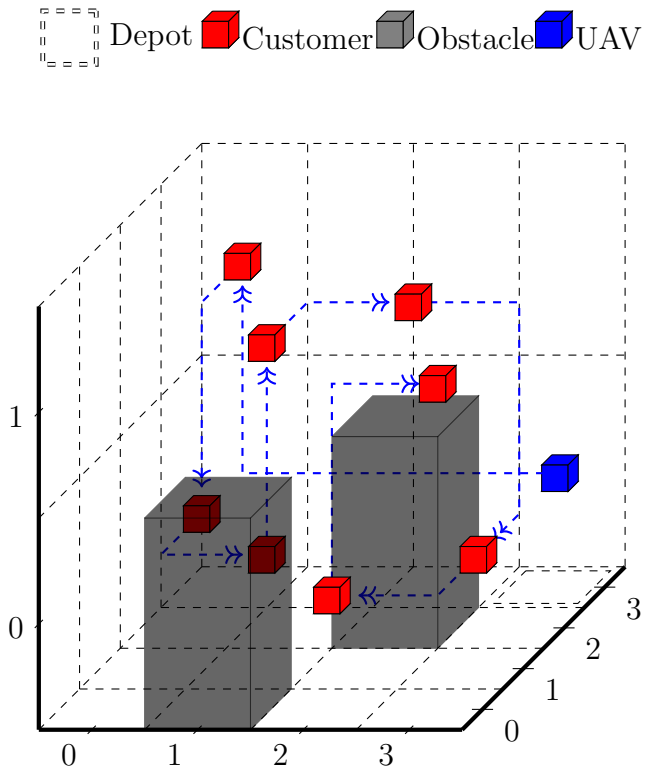
- The first type of experiment aims to measure how much energy our techniques

can save. To this end, we need to measure and compare the real energy consumption of the drone with and without taking account the presence of the wind. In other words, we need to do three different experiments:

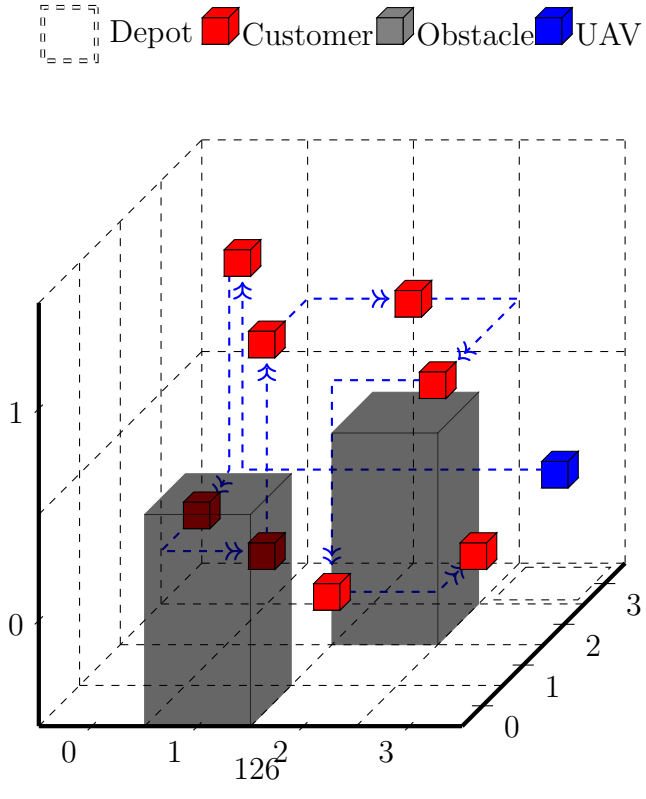
- In the first experiment, we disable the disturbance function (i.e., returning $(0, 0, 0)$), run Algorithm 9, get the plan, have the drone execute it, and measure the consumed energy. Figure 5.8a shows the flight path of the drone in this scenario, and the total energy consumption from the real experiment is $87kJ$.
- In the second experiment, we also disable the disturbance function utilized the same path, but deployed the fan at the bottom left corner of the room to continually generate a wind of $1m/s$. In this case, the drone consumed 102kJ (approximately 17% extra energy) to finish its mission.
- In the third one, we enable the disturbance function while calculating the plan. Figure 5.8b shows the flight path of the drone in this scenario, and the total energy consumption from the real experiment is $95kJ$ (approximately 9% extra energy). Note that we also actually employed the fan at the bottom left corner of the room to continually generate a wind of $1m/s$ during the real experiment.

- The second type of experiment is to show that our online algorithm can effectively react to uncertainties caused by the physical environment. We deployed two UAVs flying at two different altitudes in the working area. Due to safety and collision avoidance, we set that the blue UAV can only service customers

at the layer 0 (i.e., $z_c = 0, \forall c \in V_C$) while the green one can service only customers at the layer 1 (i.e., $z_c = 1, \forall c \in V_C$.) Moreover, although the UAVs are equipped with batteries of $200kJ$, we constrain that each of their flights (since taking off until landing) cannot consume more than $75kJ$. When the UAVs follow the flight paths prescribed by Algorithm 9 (see Figure 5.8c), the UAV flying in a lower altitude cannot complete one of its flight due to reaching their energy constraint ($75kJ$) too quickly, resulting in a pre-mature completion of the mission. Recall that when the two UAVs fly close together, the lower UAV has to spend about 12% more energy to resist the spiral air flow generated by the higher UAV. Next, we ran our online algorithm initialized by the same setting. The online algorithm effectively conducted appropriate planning and serviced all customers (see Figure 5.8d).

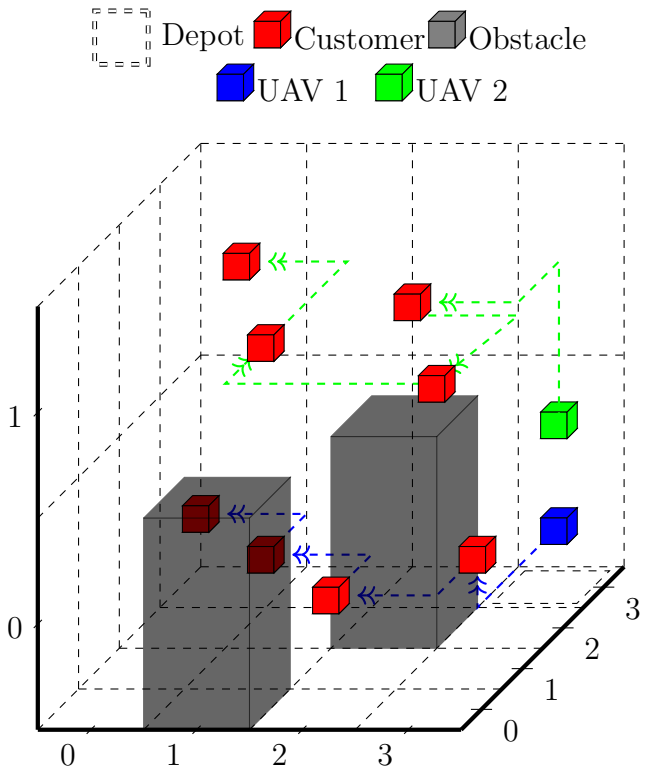


(a) Disabled disturbance function.

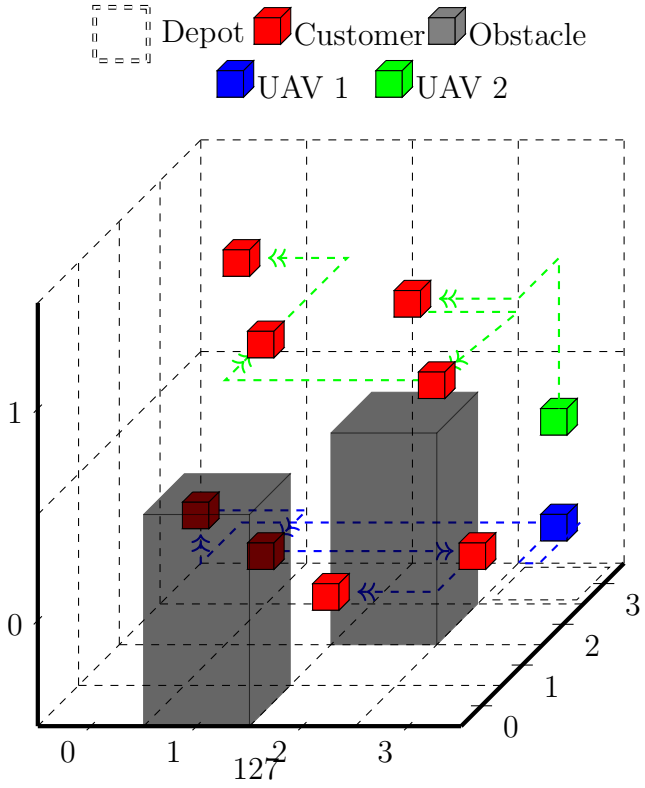


(b) Wind speed of $1m/s$

Figure 5.8: Experiments with Intel Aero UAVs.



(c) Offline planning for two UAVs.



(d) Enabled online planning

Figure 5.8: Experiments with Intel Aero UAVs (cont.).

Chapter 6

Conclusion

This thesis has tackled the problem of optimizing energy for a team of autonomous mobile robots. Our goal is to show that *exclusively software-based approaches can provide an effective energy-optimization for DAMR systems.* By effective, we mean that our systems can be actually implemented and deployed in real-world scenarios. To this end, we studied a class of DAMR systems where the team of battery-powered and networked robots navigating in a physical environment and acting in concert to accomplish a common goal. In this chapter, we present summaries and contributions of each chapter in Section 6.1 and the future work in Section 6.2

6.1 Summary

In Chapter 3, we proposed the Reliability-Energy Tradeoff problem which aims to control energy consumption in order to improve the level of reliability of our DAMR systems. To this end, we made the following contributions:

- We proposed a general-purpose model called *verificable task graphs* that capture

the interactions of the reliability vs. energy and allow system designers to parameterize the tradeoffs in their system. That is, verification tasks can be intermittently and arbitrarily inserted into the computation task graph.

- We also introduced three different offline techniques that identify optimal and near-optimal schedules for execution units to compute functional tasks as well as *peer-verification* tasks, where units verify the sanity of the output of each other to provide resilience against cyberattacks.

- We also introduced an online algorithm that reacts to situations, where the physical environment invalidates some of the design, implementation, or optimization assumptions.

- We reported results of simulations of all of our algorithms as well as a proof of concept using a real network of UAVs that carry our a joint search mission.

In Chapter 4, we introduced the problem of extending operational time of team of mobile robots by charging stations. We assume that the robots are heterogeneous (having different energy limits and being able to service different types of customers) and have access to a priori known map of the environment. The map is modeled as a directed, connected, and finite graph whose nodes are charging stations or customers, and arcs denote the possibility of traveling. Our ultimate goal is to find an optimal task assignment and path planning for the robots that minimizes energy consumption as well as the time needed to complete the tasks, including the time spent for recharging. To this end, we made the following contributions:

- We transform our optimization problem into *multi-objective integer programming* (MIP). This techniques produces the optimal result, though it is offline

and limited to small problem sizes.

- We also introduce three scalable offline algorithms that find sub-optimal solutions. The first is a *semigreedy* algorithm which is a combination of the Nearest Neighbor Heuristic and an extension of the randomized $A^\star$ algorithm. We also propose two *genetic algorithms* (GAs) named *Greedy GA* and *random GA*. The two algorithms have the same evolving (mating, mutating and selecting) operations. For generating their initial population, the Greedy GA invokes the semigreedy algorithm, while the Random GA randomly assigns customers to robots.

- We also propose an *online* algorithm to dynamically adjust the plan according to the changes in the physical environment.

- We reported results of simulations of all of our algorithms as well as a proof of concept using a real network of UAVs that carry our a joint search mission.

In Chapter 5, we propose the *Optimal Path Planning in the Present of Disturbances* problem which aims to design optimal paths for a team of mobile robots in the presence of disturbances. We assume that the robots are homogeneous and have access to a priori known map of the environment, which may contain obstacles. The map is then modeled as an undirected, connected, and finite graph whose nodes are customers, depots, and crossroads (which do not require any service from robots), and edges denote the possibility of traveling. Our ultimate goal is to find an optimal task assignment and path planning for the robots that minimizes energy consumption. To this end, we made the following contributions:

- We transform our optimization problem into *integer linear programming* (ILP). This technique produces the optimal result, though it is offline and limited to small problem sizes.

- We introduce two heuristic algorithms based on Nearest Neighbor Heuristic to find sub-optimal solutions. In the first algorithm, assignments for execution units are sequentially constructed, so it will utilizes less number execution units if possible. We also propose the second heuristic that aims to construct the missions simultaneously and is good for parallelism.

- Finally, we also propose a *online decentralized* algorithm based on negotiation.

6.2 Future Work

The thesis has tackled the problem of optimizing energy consumption for a team of autonomous mobile robots from three distinct aspects. It will be interesting to further expand this study to:

- **Dynamic number of robots.** In our thesis, we had fixed the number of mobile robots that initially take part in the problem. It would be interesting to find the exact number of robots required to produce the optimal solution to service a given graph. Moreover, although our online algorithms cover the situation where one or more robots become unavailable (i.e., the number of robots is reduced on the fly), it would be more practical if the algorithms also allows new robots join the team on the fly.

- **Dynamic graph.** We solved the problem for fixed graphs. In real life scenarios, certain customers tend to have sudden demands which would be better served in the present instance of the problem. In this case, the graphs are not known fully and robots know new demands only after reaching a location. One can design a probabilistic distributed algorithm to solve this problem.

- **Fault-tolerance.** We are interested in investigate a more complicated scenario where execution units may become *Byzantine*, i.e., they non-intentionally or maliciously misrepresent their observation or peer-verification results.

Bibliography

Ahmad, I. and Kwok, Y.-K. (1998). On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel & Distributed Systems*, (9), 872–892.

Ahmadi, M. and Stone, P. (2006). A multi-robot system for continuous area sweeping tasks. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 1724–1729. IEEE.

Amazon.com, Inc. (2016). `https://www.amazon.com/Amazon-Prime-Air/b?ie=UTF8&node=8037720011`. Accessed:2019-01-02.

Baxter, J. L., Burke, E., Garibaldi, J. M., and Norman, M. (2007). Multi-robot search and rescue: A potential field based approach. In *Autonomous robots and agents*, pages 9–16. Springer.

Bellmore, M. and Nemhauser, G. L. (1968). The traveling salesman problem: a survey. *Operations Research*, **16**(3), 538–558.

Bezzo, N., Mohta, K., Nowzari, C., Lee, I., Kumar, V., and Pappas, G. (2016). Online planning for energy-efficient and disturbance-aware uav operations. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5027–5033. IEEE.

Borndörfer, R., Schenker, S., Skutella, M., and Strunk, T. (2016). Polyscip. In G.-M. Greuel, T. Koch, P. Paule, and A. Sommese, editors, *Mathematical Software - ICMS 2016, 5th International Conference, Berlin, Germany, July 11-14, 2016, Proceedings*, volume 9725, pages 259 – 264.

Bullo, F., Frazzoli, E., Pavone, M., Savla, K., and Smith, S. L. (2011). Dynamic vehicle routing for robotic systems. *Proceedings of the IEEE*, **99**(9), 1482–1504.

Burgard, W., Moors, M., Fox, D., Simmons, R., and Thrun, S. (2000). Collaborative multi-robot exploration. In *ICRA*, pages 476–481.

Burgard, W., Moors, M., Stachniss, C., and Schneider, F. E. (2005). Coordinated multi-robot exploration. *IEEE Transactions on robotics*, **21**(3), 376–386.

Calisi, D., Farinelli, A., Iocchi, L., and Nardi, D. (2007). Multi-objective exploration and search for autonomous rescue robots. *Journal of Field Robotics*, **24**(8-9), 763–777.

Cáp, M., Novák, P., Seleckỳ, M., Faigl, J., and Vokffnek, J. (2013). Asynchronous decentralized prioritized planning for coordination in multi-robot system. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 3822–3829. IEEE.

Cardenas, A., Amin, S., Sinopoli, B., Giani, A., Perrig, A., Sastry, S., *et al.* (2009). Challenges for securing cyber physical systems. In *Workshop on future directions in cyber-physical systems security*, volume 5.

Conrad, R. G. and Figliozzi, M. A. (2011). The recharging vehicle routing problem. In *Proceedings of the 2011 industrial engineering research conference*, pages 1–8.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

Cornuéjols, G., Fonlupt, J., and Naddef, D. (1985). The traveling salesman problem on a graph and some related integer polyhedra. *Mathematical programming*, **33**(1), 1–27.

Dantzig, G. B. and Ramser, J. H. (1959). The truck dispatching problem. *Management science*, **6**(1), 80–91.

Deb, K. (2014). Multi-objective optimization. In *Search methodologies*, pages 403–449. Springer.

Desai, A., Saha, I., Yang, J., Qadeer, S., and Seshia, S. A. (2017). Drona: A framework for safe distributed mobile robotics. In *Proceedings of the 8th International Conference on Cyber-Physical Systems*, pages 239–248. ACM.

DHL International GmbH (2016). `http://www.dhl.com/en/press/releases/releases_2014/group/dhl_parcelcopter_launches_initial_operations_for_research_purposes.html`. Accessed:2019-01-02.

Di Franco, C. and Buttazzo, G. (2016). Coverage path planning for uavs photogrammetry with energy and resolution constraints. *Journal of Intelligent & Robotic Systems*, **83**(3-4), 445–462.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, **1**(1), 269–271.

Donald, B., Gariepy, L., and Rus, D. (2000). Distributed manipulation of multiple objects using ropes. In *Proceedings 2000 ICRA. Millennium Conference. IEEE*

International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065), volume 1, pages 450–457. IEEE.

El-Rewini, H. and Lewis, T. G. (1990). Scheduling parallel program tasks onto arbitrary target machines. *Journal of parallel and Distributed Computing*, **9**(2), 138–153.

El-Rewini, H., Ali, H. H., and Lewis, T. (1995). Task scheduling in multiprocessing systems. *Computer*, **28**(12), 27–37.

Erdmann, M. and Lozano-Perez, T. (1987). On multiple moving objects. *Algorithmica*, **2**(1-4), 477.

Erdoğan, S. and Miller-Hooks, E. (2012). A green vehicle routing problem. *Transportation Research Part E: Logistics and Transportation Review*, **48**(1), 100–114.

Escherich, R., Ledendecker, I., Schmal, C., Kuhls, B., Grothe, C., and Scharberth, F. (2009). She: Secure hardware extension—functional specification, version 1.1. *Hersteller-Initiative Software (HIS) AK Security*.

Fleischmann, B. (1985). A cutting plane procedure for the travelling salesman problem on road networks. *European Journal of Operational Research*, **21**(3), 307–317.

Fovino, I. N., Carcano, A., Masera, M., and Trombetta, A. (2009). Design and implementation of a secure modbus protocol. In *International conference on critical infrastructure protection*, pages 83–96. Springer.

Gatteschi, V., Lamberti, F., Paravati, G., Sanna, A., Demartini, C., Lisanti, A., and Venezia, G. (2015). New frontiers of delivery services using drones: A prototype

system exploiting a quadcopter for autonomous drug shipments. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 920–927. IEEE.

Guo, Y. and Parker, L. E. (2002). A distributed and optimal motion planning approach for multiple mobile robots. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 3, pages 2612–2619. IEEE.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, **4**(2), 100–107.

Hartuv, E., Agmon, N., and Kraus, S. (2018). Scheduling spare drones for persistent task performance under energy constraints. In *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 532–540. International Foundation for Autonomous Agents and Multiagent Systems.

Humayed, A., Lin, J., Li, F., and Luo, B. (2017). Cyber-physical systems security - a survey. *IEEE Internet of Things Journal*, **4**(6), 1802–1831.

Hwang, J.-J., Chow, Y.-C., Anger, F. D., and Lee, C.-Y. (1989). Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, **18**(2), 244–257.

Intel Aero RTF Drone (2016). `https://click.intel.com/intel-aero-ready-to-fly-drone-2702.html`. Accessed:2019-01-02.

Interian, R. and Ribeiro, C. C. (2017). A grasp heuristic using path-relinking and

restarts for the steiner traveling salesman problem. *International Transactions in Operational Research*, **24**(6), 1307–1323.

Jovanov, I. and Pajic, M. (2017). Sporadic data integrity for secure state estimation. In *2017 IEEE 56th Annual Conference on Decision and Control (CDC)*, pages 163–169. IEEE.

Jovanov, I. and Pajic, M. (2019). Relaxing integrity requirements for attack-resilient cyber-physical systems. *IEEE Transactions on Automatic Control*.

Kahn, A. B. (1962). Topological sorting of large networks. *Communications of the ACM*, **5**(11), 558–562.

Khatib, O., Yokoi, K., Chang, K., Ruspini, D., Holmberg, R., and Casal, A. (1996). Vehicle/arm coordination and multiple mobile manipulator decentralized cooperation. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems. IROS'96*, volume 2, pages 546–553. IEEE.

Kim, J. and Morrison, J. R. (2014). On the concerted design and scheduling of multiple resources for persistent uav operations. *Journal of Intelligent & Robotic Systems*, **74**(1-2), 479–498.

Kim, J., Song, B. D., and Morrison, J. R. (2013). On the scheduling of systems of uavs and fuel service stations for long-term mission fulfillment. *Journal of Intelligent & Robotic Systems*, pages 1–13.

Kolling, A. and Carpin, S. (2006). Multirobot cooperation for surveillance of multiple moving targets-a new behavioral approach. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 1311–1316. IEEE.

Kolling, A. and Carpin, S. (2008). Multi-robot surveillance: an improved algorithm for the graph-clear problem. In *2008 IEEE International Conference on Robotics and Automation*, pages 2360–2365. IEEE.

Krishnan, A., Markov, M., and Bonakdarpour, B. (2017). Distributed vehicle routing approximation. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 503–512. IEEE.

Kruatrachue, B. and Lewis, T. (1988). Grain size determination for parallel processing. *IEEE software*, **5**(1), 23–32.

Kuffner Jr, J. J. and LaValle, S. M. (2000). Rrt-connect: An efficient approach to single-query path planning. In *ICRA*, volume 2.

Kumar, S. N. and Panneerselvam, R. (2012). A survey on the vehicle routing problem and its variants. *Intelligent Information Management*, **4**(03), 66.

Kwok, Y.-K. and Ahmad, I. (1996). Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE transactions on parallel and distributed systems*, **7**(5), 506–521.

Lesi, V., Jovanov, I., and Pajic, M. (2017). Security-aware scheduling of embedded control tasks. *ACM Trans. Embed. Comput. Syst.*, **16**(5s), 188:1–188:21.

Letchford, A. N., Nasiri, S. D., and Theis, D. O. (2013). Compact formulations of the steiner traveling salesman problem and related problems. *European Journal of Operational Research*, **228**(1), 83–92.

Lin, Y. and Mitra, S. (2015). Starl: Towards a unified framework for programming, simulating and verifying distributed robotic systems. In *ACM SIGPLAN Notices*, volume 50, page 9. ACM.

Macwan, A., Vilela, J., Nejat, G., and Benhabib, B. (2015). A multirobot path-planning strategy for autonomous wilderness search and rescue. *IEEE transactions on cybernetics*, **45**(9), 1784–1797.

Mandow, L., De la Cruz, J. P., *et al.* (2005). A new approach to multiobjective a* search. In *IJCAI*, volume 8.

McFarland, M. (2016). `https://money.cnn.com/2016/09/08/technology/google-drone-chipotle-burrito/index.html`. Accessed:2019-01-02.

Miller, C. E., Tucker, A. W., and Zemlin, R. A. (1960). Integer programming formulation of traveling salesman problems. *Journal of the ACM (JACM)*, **7**(4), 326–329.

Mitchell, R. and Chen, I.-R. (2014). A survey of intrusion detection techniques for cyber-physical systems. *ACM Computing Surveys (CSUR)*, **46**(4), 55.

Morbidi, F., Cano, R., and Lara, D. (2016). Minimum-energy path generation for a quadrotor uav. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1492–1498. IEEE.

Park, H. and Morrison, J. R. (2014). On the resources required to provide persistent robotic service: Multiple immobile customers and a single service station. In

Proceedings of the Asia Pacific Industrial Engineering and Management Systems Conference (APIEMS'14).

Parker, L. E. (1999). Cooperative robotics for multi-target observation. *Intelligent Automation & Soft Computing*, **5**(1), 5–19.

Pavone, M., Frazzoli, E., and Bullo, F. (2010). Adaptive and distributed algorithms for vehicle routing in a stochastic and dynamic environment. *IEEE Transactions on automatic control*, **56**(6), 1259–1274.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, **12**, 2825–2830.

Rekleitis, I., Dudek, G., and Milios, E. (2001). Multi-robot collaboration for robust exploration. *Annals of Mathematics and Artificial Intelligence*, **31**(1-4), 7–40.

Rus, D., Donald, B., and Jennings, J. (1995). Moving furniture with teams of autonomous robots. In *Proceedings 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots*, volume 1, pages 235–242. IEEE.

Rushanan, M., Rubin, A. D., Kune, D. F., and Swanson, C. M. (2014). Sok: Security and privacy in implantable medical devices and body area networks. In *2014 IEEE Symposium on Security and Privacy (SP)*, pages 524–539. IEEE.

Saha, I., Ramaithitima, R., Kumar, V., Pappas, G. J., and Seshia, S. A. (2014). Automated composition of motion primitives for multi-robot systems from safe ltl specifications. In *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*, pages 1525–1532. IEEE.

Saha, I., Ramaithitima, R., Kumar, V., Pappas, G. J., and Seshia, S. A. (2016). Implan: scalable incremental motion planning for multi-robot systems. In *Cyber-Physical Systems (ICCPS), 2016 ACM/IEEE 7th International Conference on*, pages 1–10. IEEE.

Schneider, M., Stenger, A., and Goeke, D. (2014). The electric vehicle-routing problem with time windows and recharging stations. *Transportation Science*, **48**(4), 500–520.

Sha, L., Gopalakrishnan, S., Liu, X., and Wang, Q. (2008). Cyber-physical systems: A new frontier. In *Sensor Networks, Ubiquitous and Trustworthy Computing, 2008. SUTC'08. IEEE International Conference on*, pages 1–9. IEEE.

Sih, G. C. and Lee, E. A. (1993). A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE transactions on Parallel and Distributed systems*, **4**(2), 175–187.

Simmons, R., Apfelbaum, D., Burgard, W., Fox, D., Moors, M., Thrun, S., and Younes, H. (2000). Coordination for multi-robot exploration and mapping. In *Aaai/Iaai*, pages 852–858.

Simon, D. (2018). `https://www.cnn.com/2018/12/04/us/nypd-drones/index.html`. Accessed:2019-01-02.

Song, B. D., Kim, J., Kim, J., Park, H., Morrison, J. R., and Shim, D. H. (2014a). Persistent uav service: an improved scheduling formulation and prototypes of system components. *Journal of Intelligent & Robotic Systems*, **74**(1-2), 221–232.

Song, B. D., Kim, J., and Morrison, J. R. (2014b). Towards real time scheduling for persistent uav service: A rolling horizon milp approach, rhta and the stah heuristic. In *Unmanned Aircraft Systems (ICUAS), 2014 International Conference on*, pages 506–515. IEEE.

Stilwell, D. J. and Bay, J. S. (1993). Toward the development of a material transport system using swarms of ant-like robots. In *[1993] Proceedings IEEE International Conference on Robotics and Automation*, pages 766–771. IEEE.

Sugar, T. and Kumar, V. (2000). Control and coordination of multiple mobile robots in manipulation and material handling tasks. In *Experimental Robotics VI*, pages 15–24. Springer.

Tesla, Inc. (2008). `https://www.tesla.com`. Accessed:2019-01-02.

Turpin, M., Michael, N., and Kumar, V. (2014). Capt: Concurrent assignment and planning of trajectories for multiple robots. *The International Journal of Robotics Research*, **33**(1), 98–112.

Uber Technologies, Inc. (2009). `https://www.uber.com`. Accessed:2019-01-02.

Van Den Berg, J. P. and Overmars, M. H. (2005). Prioritized motion planning for multiple robots. In *Intelligent Robots and Systems, 2005.(IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 430–435. IEEE.

Velagapudi, P., Sycara, K., and Scerri, P. (2010). Decentralized prioritized planning in large multirobot teams. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 4603–4609. IEEE.

Wang, L., Siegel, H. J., Roychowdhury, V. P., and Maciejewski, A. A. (1997). Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach. *Journal of parallel and distributed computing*, **47**(1), 8–22.

Wang, Z., Kimura, Y., Takahashi, T., and Nakano, E. (2000). A control method of a multiple non-holonomic robot system for cooperative object transportation. In *Distributed autonomous robotic systems 4*, pages 447–456. Springer.

Waymo LLC (2009). `https://waymo.com/`. Accessed:2019-01-02.

Wing LLC (2019). `https://spectrum.ieee.org/automaton/robotics/drones/wing-officially-launches-australian-drone-delivery-service`. Accessed:2019-04-12.

Wolf, M. and Gendrullis, T. (2011). Design, implementation, and evaluation of a vehicular hardware security module. In *International Conference on Information Security and Cryptology*, pages 302–318. Springer.

Wu, A. S., Yu, H., Jin, S., Lin, K.-C., and Schiavone, G. (2004). An incremental genetic algorithm approach to multiprocessor scheduling. *IEEE Transactions on parallel and distributed systems*, **15**(9), 824–834.

Wu, M.-Y. and Gajski, D. D. (1990). Hypertool: A programming aid for message-passing systems. *IEEE transactions on parallel and distributed systems*, **1**(3), 330–343.

Xia, F., Ma, L., Dong, J., and Sun, Y. (2008). Network qos management in cyber-physical systems. In *Embedded Software and Systems Symposia, 2008. ICESS Symposia'08. International Conference on*, pages 302–307. IEEE.

Zeng, Y. and Zhang, R. (2017). Energy-efficient uav communication with trajectory optimization. *IEEE Transactions on Wireless Communications*, **16**(6), 3747–3760.

Zimmerman, A. (2013). Starl for programming reliable robotic networks.