2020-08-07

A Framework for Simulating and Analyzing Multi-UAV Persistent Search and Retrieval with Stochastic Target Appearance

Ryan David Day
Brigham Young University

A Framework for Simulating and Analyzing Multi-UAV Persistent Search and Retrieval with Stochastic Target Appearance

Ryan David Day

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Mechanical Engineering

Brigham Young University

ABSTRACT

A Framework for Simulating and Analyzing Multi-UAV Persistent Search and Retrieval with Stochastic Target Appearance

Ryan David Day
Department of Mechanical Engineering, BYU
Master of Science

In recent years, advances in small unmanned aerial vehicle (UAV) technology have transformed the use cases of these aircraft from hobby flying to industrial and business applications. These maneuverable, easily deployed tools can be retrofitted with a myriad of sensors and equipment, which make them suitable to perform a variety of specialized tasks. With increasing UAV capabilities, the function of small UAVs can be extended from pure monitoring or surveillance to the dual objective of monitoring an environment for events and addressing the events in some way. This thesis seeks to explore a subdomain of the dual objective problem described, referred to in this thesis as the multi-UAV persistent search and retrieval task with stochastic target appearance (PSR-STA), in which UAVs continuously search an area over a long period of time for targets of interest, which appear according to a probabilistic model, to retrieve and deliver them to a collector location.

The advent of high-speed computers and agent-based modeling theory enable the simulation of multi-UAV PSR-STA. However, it can be complicated to combine parts of multi-UAV PSR-STA such as motion models and multi-UAV coordination into one integrated system, and even after they are combined successfully, it is difficult to analyze the system except with simple comparison tools. This thesis 1) proposes a framework that builds a foundation for understanding how to simulate and analyze multi-UAV PSR-STA through prescribing important design decisions and methods for simulation and 2) identifies metrics, analysis tools, and trends related to overall system effectiveness for multi-UAV PSR-STA.

A case study of multi-UAV park cleanup is implemented where many simulations with input parameters chosen by a latin hypercube design of experiments are examined, algorithms for choosing the locations of collectors and charging stations based on probabilistic information are proposed, and the differences in effectiveness between four coverage search patterns are analyzed. Measures are highlighted that provide insight into performance variability over time and space. Line charts and the discrete Fourier transform are used to understand temporal patterns inherent in the data. Principal component analysis is used to analyze relevant spatial patterns in effectiveness, and a random forest surrogate model with a profiler is used to explore the non-linear influence of input parameters on the spatial patterns. The trellis chart or figure of figures method is presented for visualizing spatial and temporal data across many simulations. A second set of experiments based on the park cleanup case study are performed and examined to verify the benefits of these methods.

Keywords: multi-UAV search, UAV target retrieval, spatial analysis, temporal analysis

ACKNOWLEDGMENTS

First and foremost I would like to thank my wife, Catharine, for her continual support and optimism throughout the thesis writing process. Her friendship and companionship has been invaluable to me, impacting all areas of my life. I cannot express enough how much she means to me. I would like to thank my parents for encouraging my curiosity and helping me develop a love of learning, which led me to pursue a degree in a challenging field and enjoy the process. My aunt and uncle, Judy and Justin, and their family have also been a great support as I have worked to write this thesis.

Dr. John Salmon has been a great mentor and friend. His love of new ideas and mastery of systems engineering has always inspired me. Without him I would not have this wonderful opportunity to pursue a master's degree. His encouragement and effort to help me achieve the goal of completing this thesis has been invaluable to my progression as an engineer, and his kind and hard working nature has made the experience one I will never forget.

Dr. Tim McLain and Dr. Cammy Peterson have been a great help to me. Dr. McLain was the professor for two of the most challenging and useful classes I took at BYU, design of control systems, and autonomous systems, and his guidance in completing my thesis is much appreciated. Dr. Peterson has always been so willing to meet with me and help me with any challenges or questions that I have proposed to her.

Finally, my peers in my research lab have impacted my life for the better, including Carsten Christensen and Landon Willey, who have been great friends and counselors throughout my time as a research assistant, Spencer Bunnell, whose impassioned discussions about predictive modeling and general engineering principles will always be remembered, and Jon Sadler, who helped me build a foundation of good coding practices that led me to become a proficient software engineer.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

$\mathcal{P}$	Park
$l_{\mathcal{P}}$	Park side length
$l_{\mathcal{A}}$	Area side length
γ	Expected value for binomial distribution related to target appearance
N_C	Number of collectors
N_R	Number of charging stations
s	UAV speed
T_F	UAV flight time
T_R	UAV recharge time
τ_c	Time delay representing a UAV landing at a charging station
τ_{to}	Time delay representing a UAV taking off from a charging station
τ_{rt}	Time delay representing a UAV retrieving trash
τ_{dt}	Time delay representing a UAV depositing trash in a collector
r_d	UAV detection radius
$d_{UAV,t}$	Distance from a UAV to a trash
$d_{t,c}$	Distance from a trash to a collector
$d_{c,r}$	Distance from a collector to a charger
C_1	Constant factor accounting for uncertainty
C_2	Constant factor accounting for uncertainty
$d_{UAV,r}$	Distance from a UAV to a charger
$d(c,v)$	Distance between the centroid and a vertex of a polygon
$d_{\perp_{max}}$	Longest perpendicular distance between the longest edge of a polygon and any of its vertices
d_{edge}	Constant that represents the distance from the edges of a polygon to the search pattern
d_{lane}	Constant that represents the distance between lanes in the lawnmower pattern
C_3	Constant factor that adjusts d_{edge} to make sure the whole space is covered
$\mathcal{M}$	Set of positions
N_p	Number of positions in $\mathcal{M}$
d_{avg}	Average distance from any point to its closest position in $\mathcal{M}$
$A_{\mathcal{P}}$	Area of a polygon
$w(x,y)$	Weighting function with (x, y) being cartesian coordinate inputs
$\mathcal{G}$	Set of square grid cells of equal area
$\mathcal{G}_i$	Set of square grid cells at time step i
r_g	UAV detection radius in number of grid cells
l_g	Length of a grid cell
T_S	Simulation run time
$\mathcal{Q}$	Set of trash that appeared in the simulation over all time steps
$\overline{T}_r$	Average time of trash retrieval
T_r^t	The amount of time from the appearance of trash t to its retrieval by a UAV
$\overline{N}_t$	The average number of trash left out at each time step
$\mathcal{Q}_i$	Set of trash left out at time step i
$\overline{T}_v$	The average time any area in the simulation was last searched
t_{LS}	Time last searched
$\overline{t_{LS}}$	Average time last searched

$\mathcal{T}_i$	Set of all targets at time step i
t_t	Amount of time a target t has been present in the simulation since appearing

CHAPTER 1. INTRODUCTION

1.1 Problem Definition and Research Statement

In recent years, advances in small unmanned aerial vehicle (UAV) technology have transformed the use cases of these aircraft from hobby flying to industrial and business applications. These maneuverable, easily deployed tools can be retrofitted with a myriad of sensors and equipment, which make them suitable to perform a variety of specialized tasks. Many UAV applications relate to persistent monitoring or searching, which involve the UAV flying through an area, using a video camera or other sensor devices mounted on the UAV to detect changes in an environment or to search for objects of interest. Some real-world examples of these applications include search and rescue [1], building inspection [2], and military surveillance [3].

With increasing UAV capabilities, the function of small UAVs can be extended from pure monitoring or surveillance to the dual objective of monitoring an environment for events and addressing the events in some way. Examples of this are graffiti removal [4], where UAVs must search a city for graffiti and paint over it, and pesticide application [5], where UAVs must apply pesticide to pest-infested areas of an agricultural environment. These activities often include events that appear according to some probabilistic pattern, such as pests appearing more frequently in certain areas of a field of crops, or graffiti being more likely to be created in specific parts of a city. The addition of addressing events after locating them adds a new layer of complexity to UAV search, and new methods must be developed to simulate and analyze these kinds of scenarios.

This thesis seeks to explore a subdomain of the dual objective problem described, referred to in this thesis as the multi-UAV persistent search and retrieval task with stochastic target appearance (PSR-STA), in which UAVs continuously search an area over a long period of time for targets of interest, which appear according to a probabilistic model, to retrieve and deliver them to a collector location. This task is an extension of the persistent surveillance task, in which UAVs persistently monitor a known environment [6], with the search and retrieval task, where agents must find targets in an area and deliver them to a predefined location [7]. The surveillance task is extended by including stochastically appearing targets that must be retrieved and delivered to a collector location upon discovery. An example of an application that motivates the study of multi-UAV PSR-STA is litter removal, where litter is discarded by people in an area [8] and retrieved and deposited into a trash bin by a UAV or other autonomous agent [9,10]. A study prepared for the Environmental Protection Agency estimated that west coast communities in the United States of America spend more than $520,000,000 each year to combat littering, and hundreds of species of animals are affected as the litter is eventually displaced to the ocean [11]. This emphasizes the need for studying and understanding multi-UAV PSR-STA for successful deployment of UAVs to help with this task, as their low cost and ability to interact with the environment without an operator would help to improve communities and reduce cost through autonomous litter collection.

Extending persistent surveillance with the search and retrieval task reveals rich and exciting research considerations that should be explored to design solutions for a given scenario. First, UAV autonomy must be considered. This includes analyzing coordinated multi-UAV search strategies and determining methods to enable persistent UAV operation beyond the battery life of an individual UAV. Second, decisions regarding the number and locations of battery recharging stations to aid persistent operation, and the number and locations of collectors to facilitate effective target retrieval strategies must be considered. Previous work has explored arbitrary numbers and locations for recharging stations [12,13], or optimized a chosen number of recharging locations for tasks without stochastic elements [14,15]. Multi-UAV PSR-STA motivates an augmentation of these methods to design a collector and charger placement algorithm based on stochastic event information.

Metrics of interest must be defined and measured in a wide range of circumstances to gain insight into how parameters influence system effectiveness in multi-UAV PSR-STA. Since testing many variations of multi-UAV search scenarios in the real world is time and cost prohibitive, a common methodology for understanding effectiveness in situations involving UAVs is to create a computer simulation of the problem domain and run the simulation many times, varying chosen parameters while recording outputs of interest in each simulation [16–20]. After the simulation is run many times in different scenarios, potential causal and corollary relationships can be established among the varied parameter inputs and the effectiveness measures, and trends can be understood about which inputs are most influential to the output metrics. From these relationships, conclusions can be drawn about which parameters are most influential over a range of scenarios. Many who research areas related to UAV search only use one or two metrics that summarize the effectiveness of a simulation [21, 22]. This can be useful for comparing search performance of different search algorithms in a specific scenario, where few input parameters are varied, and many insights into search algorithm performance can be gained from this approach. However, if these patterns are to be implemented in real-world situations, the search algorithms may need to be deployed in many different scenarios with non-equal area sizes and with different types of UAVs, which calls for more advanced analysis methods that take into account the multidimensional parameter inputs.

The effectiveness of the UAVs in multi-UAV PSR-STA could vary through time and space depending on the combination and levels of input parameters. Non-deterministic search behavior is present even when UAVs follow a deterministic coverage search pattern since UAVs must pause their search for a significant amount of time when both retrieving targets and delivering targets to a collector location. The result of this non-deterministic variance is that if one were to only use one or two aggregate measures of effectiveness for understanding UAV search performance, information about time-based and space-based patterns present in the simulation could be obscured. Detailed analyses that reveal information about spatial and temporal variations and patterns inherent in the search behavior beyond simple quantification of effectiveness are then desirable. These analyses will aid in the understanding of important multidimensional patterns, thus allowing for the characterization of tradeoffs in the system that will inform educated decisions related to the implementation of multi-UAV PSR-STA in real-world scenarios.

The advent of high-speed computers and agent-based modeling theory enable the simulation of multi-UAV PSR-STA. However, it can be complicated to combine the different parts of persistent surveillance and search and retrieval such as motion models, battery life, and multi-UAV coordination into one integrated system. The components involved in the system may involve varying levels of assumptions that could influence the results of the simulations, and so must be carefully considered. Even when these elements are integrated and simulated successfully, it is difficult to analyze the system except with simple comparison tools. Groundwork should be laid for understanding which areas to focus on when simulating multi-UAV PSR-STA, in addition to understanding relevant metrics and methods of analysis to judge system effectiveness. Therefore, **the research objective of this thesis is to generate a framework that builds a foundation for understanding how to simulate and analyze multi-UAV PSR-STA through prescribing important design decisions and methods for simulation, and identify metrics, analysis tools, and trends related to overall system effectiveness.**

1.2 Research Outcomes

This thesis focuses on two primary areas that contribute to the accomplishment of the stated objective. The first is understanding what design decisions must be made to simulate multi-UAV PSR-STA. The first half of this thesis proposes a simulation framework that outlines these important design decisions. An analysis framework highlighting analysis areas to focus on with multi-UAV PSR-STA is also presented. A case study is implemented with the framework as a guide to demonstrate the consequences of these design decisions. As part of implementing the framework, methods are developed for collector placement and charger placement based on probabilistic information.

The second area is determining how to analyze multi-UAV PSR-STA after it has been successfully simulated, identifying important trends and patterns related to system effectiveness. To do this, metrics that quantify effectiveness are identified. Exploratory analysis methods for understanding the system behavior of a single simulation are presented, and statistical and graphical techniques comparing simulations across a wide range of scenarios are demonstrated. Furthermore, methods that allow one to identify spatial and temporal trends common across multiple simulations are presented. A second case study is performed to verify the benefits of these methods.

In summary, the outcomes of this thesis for completion of the objective are:

1. Propose a framework that facilitates simulation design through the identification of design decisions that should be made to successfully simulate multi-UAV PSR-STA

2. Implement a simulation model and necessary algorithms for successful study of multi-UAV PSR-STA, including a method for placement of chargers and collectors dependent on probabilistic information

3. Identify important metrics to characterize system effectiveness of multi-UAV PSR-STA and identify trends related to these metrics

4. Examine many different simulations of multi-UAV PSR-STA to verify the usefulness of the framework, metrics, and methods developed as a result of previous outcomes

1.3 Document Organization

The outcomes discussed in Section 1.2 are addressed in the subsequent chapters of this thesis. Chapters 2 and 3 are separate, self-contained journal articles recently submitted to peer-reviewed journals. In Chapter 2, the simulation and analysis framework is introduced. Important design decisions for simulating multi-UAV PSR-STA are presented along with related literature regarding these decisions, and general principles and simple metrics are laid out for analyzing multi-UAV PSR-STA. A case study based on a UAV park cleanup is performed, and to implement this case study, methods are demonstrated that 1) facilitate persistent UAV operation, and 2) provide a methodology for determining charger and collector placement based on probabilistic information. Four different multi-UAV search patterns are examined, and their performances are compared in different scenarios. Analysis techniques are used to spot deficiencies in search patterns and understand trade-offs in the system.

Chapter 3 extends the analysis techniques and metrics presented in Chapter 2 for the purpose of recognizing and quantifying spatiotemporal trends of overall system effectiveness. Techniques of dimensionality reduction and graphical comparison are used for understanding the temporal and spatial patterns inherent in individual simulations. Comparative analyses for a wide range of scenarios are performed using similar methods. An additional set of simulations based on the case study in Chapter 2 are performed, with the resultant data analyzed for spatiotemporal trends and patterns.

Chapter 4 discusses the conclusions of this research, as well as limitations and possible future work. Following Chapter 4 is Appendix A, which consists of the source code used for this thesis.

CHAPTER 2. A FRAMEWORK FOR MULTI-UAV PERSISTENT SEARCH AND RETRIEVAL WITH STOCHASTIC TARGET APPEARANCE IN A CONTINUOUS SPACE

2.1 Preface

This chapter introduces a framework for multi-UAV PSR-STA. Design decisions are introduced for understanding how to successfully simulate multi-UAV PSR-STA. Tools for analyzing search algorithm effectiveness through statistical and graphical methods are presented. A case study of multi-UAV park cleanup is implemented to demonstrate the framework, where algorithms for choosing the locations of collectors and charging stations based on stochastic target appearance models are proposed, methods for continuous multi-UAV operation over a long period time are demonstrated, and the differences in effectiveness between four coverage search patterns are analyzed.

2.2 Introduction

Battery powered autonomous unmanned air vehicles (UAVs) are becoming prevalent in many applications [23–25]. The primary focus of this chapter is to introduce a framework for one such application, the persistent search and retrieval task with stochastic target appearance (PSR-STA), in which UAVs intelligently and systematically search an area for stochastically appearing targets of interest to retrieve and deliver them to a collector location. This task is an extension of the persistent surveillance task, in which UAVs persistently monitor a known environment [6], with the search and retrieval task, where agents must find targets in an area and deliver them to a predefined location [7]. The surveillance task is extended by including stochastically appearing targets that must be retrieved and delivered to a collector location upon discovery. Examples of this problem domain are environmental sample collection or litter removal.

Extending persistent surveillance with the search and retrieval task reveals rich and exciting research questions that should be answered to design solutions for the problem. First, UAV autonomy must be considered. This includes choosing coordinated multi-UAV search strategies and determining methods to enable persistent UAV operation beyond the battery life of an individual UAV. Secondly, deciding the number and locations of battery recharging stations to aid persistent operation, and the number and locations of collectors to facilitate effective target retrieval strategies must be considered. Previous works have considered arbitrary numbers and locations for recharging stations [12, 13], or optimized a chosen number of recharging locations for tasks without stochastic elements [14, 15]. Multi-UAV PSR-STA motivates an augmentation of these methods to design a collector and charger placement algorithm based on stochastic event information.

The advent of computer simulations and agent-based models enable the simulation of multi-UAV PSR-STA. However, it can be complicated to combine the different parts of persistent surveillance and search and retrieval such as motion models, battery life, and multi-UAV coordination into one integrated system. Even when these elements are integrated and simulated successfully, it is difficult to analyze the system except with simple comparison tools. A framework is introduced for PSR-STA that helps facilitate simulation design and analysis. Design decisions that should be made to successfully simulate PSR-STA are introduced. Methods are described for solving challenges related to UAV autonomy, charger placement, and collector placement. Tools are presented for analyzing search algorithm effectiveness and understanding how different parameters influence the outcome of a simulation. The example of a multi-UAV park cleanup scenario is used to demonstrate the framework and show examples of how to understand and design solutions for problems related to PSR-STA. Four different multi-UAV search patterns are examined, and their performance is compared in different scenarios. Analysis techniques are used to spot deficiencies in search patterns and understand trade-offs in the system.

2.3 Related Works

2.3.1 Target Search and Retrieval

Foraging and multi-foraging, the study of agents that must find resource locations, collect them, and deposit them at a specific location, is an example of a coordinated search and retrieval task [7, 26–28]. Foraging takes place in an unknown environment and emphasizes decentralized communication schemes between agents to achieve tasks with minimal interference between agents and minimal communication [29]. The effectiveness of foraging is most influenced by information exchange and exploration vs. exploitation tradeoffs [30] since the agents do not usually share global information [31]. Though multi-foraging includes target search and retrieval, it takes place in an unknown environment, and so the problem domain focuses on individual exploration, local communication, and task allocation strategies that coalesce into effective emergent behaviors, a bottom-up approach [32]. In PSR-STA, the environment is known, which allows for centrally coordinated search patterns, a top-down approach.

Others have studied the problem of UAV cooperative autonomous search and retrieval of small objects in uneven terrain, but focus on coverage patterns for a single search in an environment [33] and complex real world implementation problems such as identifying and grasping objects [34, 35].

2.3.2 Persistent Surveillance

Persistent surveillance, also known as persistent coverage, involves visiting areas repeatedly to complete tasks or monitor changes in an environment. Many formulate these kinds of problems as repeatedly visiting waypoints [36, 37]. This transforms the problem into a variant of the traveling salesman problem (TSP), which has many heuristic solutions [38, 39]. If the problem cannot be defined as a TSP, a solution is to partition the area and assign a UAV to search each partition [40]. Since each UAV has its own partition to patrol, it is easy to deploy multiple UAVs while avoiding potential collisions. These partitions range from simple square or hexagonal grids [41] to Voronoi partitions [42]. The UAVs deploy local searching patterns in each partition [43], and additional algorithms are used to determine which partition each UAV will visit based on energy efficiency [44] and other criteria.

Modeling UAV recharging for continuous operation adds another layer of complexity to the problem. In variations of the persistent surveillance problem that include battery recharging, a common solution is to first model one or more charging stations in arbitrary locations that the UAVs repeatedly visit to recharge. The UAV search strategy is then optimized based on the charging station locations and the area of interest [12, 13, 45–48]. Other solutions simultaneously optimize charging station placement and search patterns with a genetic algorithm or a heuristic search technique [14, 15, 49, 50].

2.3.3 Persistent Surveillance Analysis Methods

Many techniques have been developed for analyzing persistent surveillance. One way is to run many different simulations, varying input parameters of interest [12, 13]. The parameter combinations can be decided with a design of experiments (DOE) methodology such as Latin Hypercube [16], or Monte Carlo Sampling [51]. During these simulation runs, outputs of interest are measured and recorded. The effect of the parameters on the outputs of interest can be understood by making a surrogate model of one of these outputs using the simulations runs, and exploring the surrogate model behavior [16]. Another analysis method is to use tools based on agent-based modeling to identify emergent behaviors in the simulations [52].

When persistent surveillance is defined as visiting discrete waypoints, connected in a graph, one metric of interest is time since each waypoint was last visited [53]. This metric can be weighted by a numerical value representing the importance of each waypoint [21]. In the case of a continuous space, the area can be divided up into grid cells each containing the value of time since last visit. When the UAV visits the cell, its time is reset to zero [6]. If the detection model is probabilistic, the cell value can be a measure related to the probability of a target existing in the cell instead of the time last visited [54].

It is often useful to compare metrics of interest graphically for analysis. Aggregate outcome parameters are often compared using bar or line charts, sometimes with confidence intervals included [55]. Heat maps can also be useful for displaying spatial data. The area of

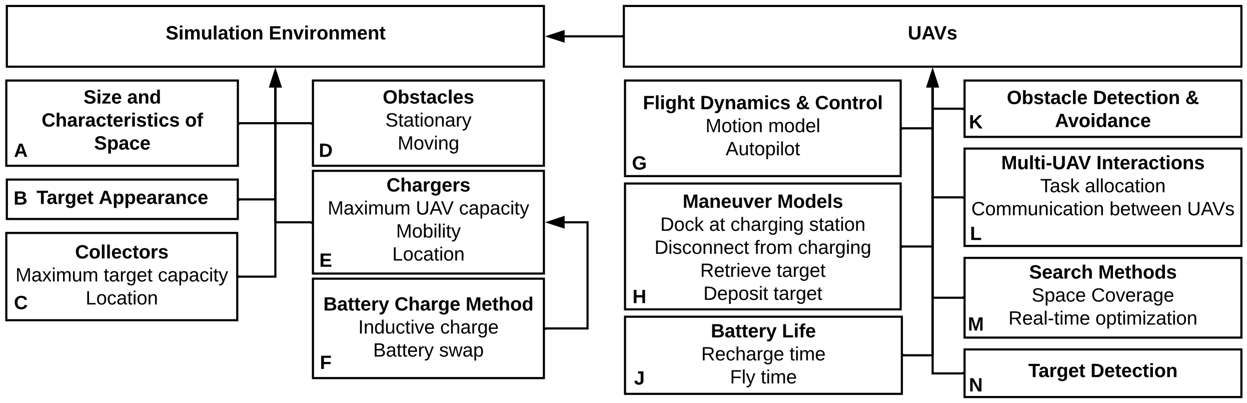


Figure 2.1: Simulation design overview, where boxes represent elements and subelements, and arrows represent subelement relationships, pointing to the parent element. Bold text is the title of each element and additional text in the boxes are potential design decisions for the respective element

interest can be divided into grid cells, each with a value representing an output at that space, represented on a color scale. Li et al. compared average visit time heat maps to compare two different search strategies [49]. These heat maps showed how one strategy visited an important area more often than another strategy. Others use 3D bar charts or surface plots to explain similar data, but these should be avoided since they can be misleading when used for comparison [56].

2.4 Methodology

This framework for the persistent search and retrieval is split into two sections: The first addresses how to design and simulate the problem domain, and the other on how to analyze it. Although decisions about the simulation design have large effects on the methods of analysis, there are some common tools that can be used regardless.

2.4.1 Simulation Design Framework Overview

An overview of the framework's design decisions that must be determined is shown in Fig. 2.1. The UAVs form an important part of the simulation environment, but require many more design decisions from other subcomponents of the environment. When different elements from Fig. 2.1 are mentioned, they are labeled with the corresponding letter in the text.

Simulation Environment Modeling

The simulation environment represents the physical space of interest for the search and retrieval problem, where targets will stochastically appear and UAVs will search (Fig 1. box A). It can be modeled by a series of vertices that form a 2D bounded polygon, or complex 3D data including elevation, terrain type, and weather conditions [57]. Obstacles can also be represented in the environment (Fig 1. box D). These can include simple stationary obstacles such as buildings or trees as well as moving obstacles such as humans, animals, or debris. These decisions can introduce complications in UAV path planning and coordination, and so are important to consider. Along with the static features, it is critical to model the process by which targets appear in the simulation (Fig 1. box B), which is important because UAVs will base their search behavior on the target appearance model [58], and determining the search behavior is a primary research question to answer, as discussed in Sect. 2.2. One way to simulate target appearance is by basing the model on time and spatial distributions [59], but data from real world scenarios can also be used to inform the model. Other features to model are chargers (Fig 1. box E) and collectors (Fig 1. box C). Collectors are locations that are designated for UAVs to deposit targets and can require a maximum capacity of targets. The charger locations, where UAVs can land and replenish their energy, can be mobile [60] or stationary [48], and can charge UAVs (Fig 1. box F) inductively [61], with a battery swapping methodology [62], or through many other methods [63]. These charger design decisions can affect the UAV charging strategy, which can ultimately influence overall system effectiveness.

UAV modeling

There are a myriad of types of UAVs that can be modeled for a multi-UAV task, but common types are based on fixed wing and multi-rotor designs [64]. UAV behavior modeling starts with the motion model (Fig 1. box G). This can range from a simple Dubins model [65] to a more complex model that matches the specific behavior of a UAV [66]. For more complicated models, autopilot, path following, and state estimation must be considered to direct the behavior of the UAV [67].

Another element of the UAV is modeling the maneuvers (Fig 1. box H), or activities performed other than simple flying between two points. There are four UAV maneuvers identified with PSR-STA: docking at a charging station, resuming flying after energy replenishment, retrieving a target, and depositing a target in a collector location. Docking at a charging station depends on the type of charging station. A battery swap station may involve a specific docking method where a system at the station swaps the battery in the UAV [62]. Inductive charging stations may require modeling how a UAV lands with an orientation on a charging pad that allows for wireless energy transfer [68]. Retrieving a target involves descending and picking up an object [69]. If the targets could be heavier than the maximum payload of the UAV, then multiple UAVs picking up a target could be modeled [70]. Depositing a target may be similar to retrieving a target and could be approximated in a similar manner to the retrieval model.

For target detection (Fig 1. box N), detailed models of camera based [71] detection can be included, as well as simpler models such as approximating sensor functionality as seeing everything in a radius. These models can depend on distance from the target, speed, attitude, altitude, and other parameters. Obstacle detection and avoidance (Fig 1. box K) is a related element to target detection, since similar detection models can be shared for detecting obstacles and targets. Obstacle avoidance can involve algorithms such as potential fields or D* [72], planning around obstacles with an online optimization algorithm at each time step [73], and cooperative obstacle avoidance [74].

Another important UAV element to model is its limited flight time based on energy capacity (Fig 1. box J). This could range from a simple linear model of flight time where there is always a constant amount of time for flying after refueling to a complex non-linear model with rates of energy depletion depending on speed, payload [75], or the maneuver being performed [76, 77].

Motion models, maneuvers, target and obstacle detection, and limited flight time all influence a critical design decision: UAV autonomous behavior. Search methods must be modeled to find stochastically appearing targets (Fig 1. box M). These can be implemented as deterministic space coverage algorithms [78] or real-time optimized search algorithms [79]. They can be informed by knowledge of a known target appearance distribution [58] or recalculated at each time step based on learned information about where targets appear [80]. Multi-UAV interaction and coordination such as task allocation between UAVs [81] and communication constraints [82] can also be considered (Fig 1. box L). All of these elements contribute to the multi-UAV PSR-STA and are important design decisions that can affect analysis.

2.4.2 Analysis Framework Overview

In any analysis framework, goals and metrics of effectiveness must be defined. With persistent surveillance, one overall goal may be to minimize the amount of time for target retrieval, or to minimize the amount of targets in the area at one time. In some cases the goal may be to keep these metrics at steady state values. Different practical implementations of the problem domain will produce variations on these goals based on factors such as noise restrictions and energy efficiency, but all will likely be related to the amount of time targets are present or the number of targets in the simulation.

Regardless, two important factors to understand are how effective the UAV autonomy strategy is for achieving a goal, and how many resources such as UAVs, chargers, and collectors it takes to service a situation with a given appearance frequency of targets. These factors both influence effectiveness, but are independent of each other. If the UAVs have a terrible search strategy, but there are many more UAVs than needed to retrieve and deposit targets, goals could be met. Likewise, UAVs could have a proven optimal search strategy, but if there are not enough UAVs to retrieve and deposit all the targets that appear, goals would not be met. In real world scenarios it is often advantageous to meet a goal with the fewest resources necessary, or to meet a goal within a budget, and so it is important to understand how UAV autonomy strategy and resource requirements influence effectiveness. DOE and statistical tests can help illuminate how parameters of interest affect goal metrics.

Different types of surrogate models such as linear regression can be employed with each simulation run as a data point to understand the practical and statistical significance of different parameters on the metrics of effectiveness. Parameter estimates can be examined, or optimization techniques can be used on these models to find optimal parameters to meet a goal. Graphical visualizations can also help to reveal patterns and understand trends including line charts, which help visualize values over time, and heat maps, which help visualize spatial patterns that can be hard to understand from simple aggregate values.

2.5 Framework Implementation

This research implements the framework with a case study of a multi-UAV park cleanup to demonstrate how to use the design framework to model a scenario, introduce algorithms that solve common problems arising in this problem domain, and present analysis tools that help to understand the effectiveness of UAV search patterns. In multi-UAV park cleanup, trash targets appear and UAVs search the park to retrieve the trash and deposit it in collector locations. The simulation environment is a square park $\mathcal{P}$, such that $\mathcal{P} \subset \mathbb{R}^2$ with origin (0, 0), side length $l_{\mathcal{P}}$ in meters. A square shape was chosen as most parks can reasonably be generalized into a combination of square shapes, and thus results from a square park can be generalized to a larger set of parks with irregular shapes. The trash targets are generated through an assumption of littering by humans, and so the amount and location of where trash appears can vary considerably depending on the park. For this case study, it was assumed that the trash stochastically appears in $\mathcal{P}$ over time according to a binomial distribution with an expected value of γ, with units of trash per hour. The location of the target is chosen with a spatially uniform random distribution inside $\mathcal{P}$ upon arrival. This simplification can be made more sophisticated with different distributions used for the arrival rate and the location of arrival, but the uniform distribution was implemented to simplify the modeling of littering while still having an adjustable parameter, γ, that influences how often trash appears in the park.

Inside $\mathcal{P}$ there are a number of collector locations N_C with unique positions, designated as places where UAVs can deposit found trash. There are also a number of charging stations N_R in $\mathcal{P}$, each with multiple inductive charging pads that the UAVs can land on to recharge. The stations are assumed to be connected to a power grid, and so have a constant supply of power. Furthermore, each station was assumed to have been set up with enough pads to charge the UAVs that landed on them for the duration of a simulation. This assumption was made because it was presumed that information about which UAVs will charge on which station is taken into account with choosing how many pads to allocate to each charging station.

UAVs are modeled as agents with a speed and a heading, as described by Dubins [65]. The UAV was assumed to be a quadcopter, with parameters based on the specifications of the DJI phantom 4 Pro. The nominal UAV speed s was set at three meters per second. The flight time T_F was set at 30 minutes. The recharge time from a depleted battery to a full battery T_R was set at one hour. There were four unique maneuvers other than searching that the UAVs had to perform to complete their tasks: 1) Docking to charge, 2) Taking off after charging, 3) Retrieving trash, and 4) Depositing trash in a collector. All of these maneuvers were modeled as constant time delays so that complicated dynamics would not have to be implemented in the simulation, since this is beyond the scope of this case study. Assuming a robust control model, landing and taking off from a charging location should take a near constant amount of time. A delay of one second was added to represent model acceleration decrease and increase from s when the UAV is landing at the charging station τ_c and taking off τ_{to}. Retrieving trash and depositing trash were modeled as 5 second delays (τ_{rt} and τ_{dt} respectively) after reaching trash and collector locations so that the modeling could be independent of any trash retrieval method such as grasping with an arm, scooping or other similar specific techniques.

Detection is often modeled as a probabilistic phenomenon [83,84]. For the case study, however, trash detection was modeled as the UAVs always being able to always detect trash within a circle centered on itself with radius r_d and could not detect trash outside this distance. This simplification was made so that non-probabilistic search patterns could be studied. Finally, the UAVs did not avoid each other, it was assumed they flew at slightly different altitudes when crossing paths, and there were no obstacles considered in the park. Therefore, no avoidance algorithms were necessary.

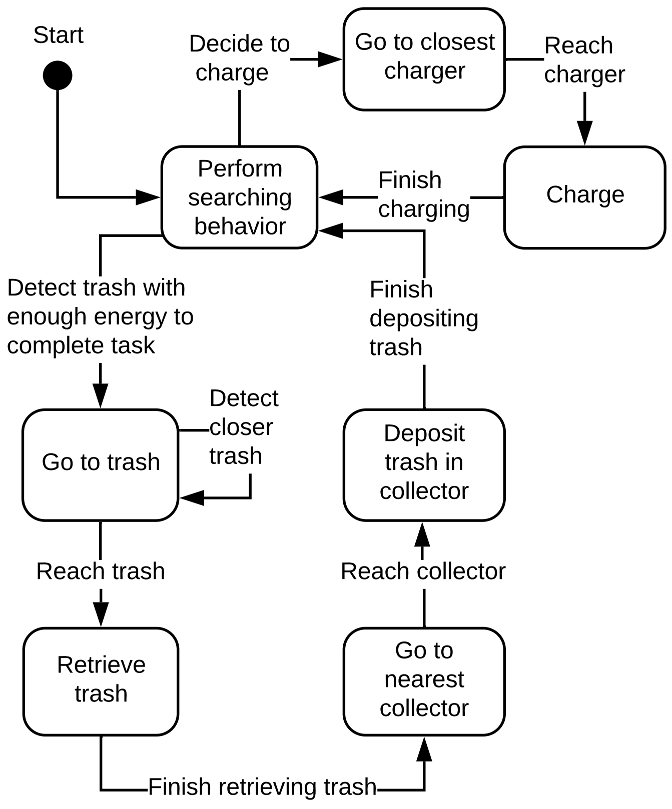


Figure 2.2: UAV autonomy state diagram for park cleanup

2.5.1 UAV Autonomy

The decision making process for each individual UAV is represented by the state diagram shown in Fig. 2.2. A UAV starts at the beginning of a simulation on an arbitrary charging pad. It takes off and immediately starts searching the park with a specific searching strategy. If the UAV sees trash less than r_d away during the search, it evaluates an inequality to see if it has enough battery power to travel to the trash, retrieve it, deposit it, and make it to a charging station if necessary. This condition is represented in Eq. 2.1, where $d_{UAV,t}$ is the distance from the UAV to the trash, $\min(d_{t,c})$ is the closest distance from the trash to any collector, $\min(d_{c,r})$ is the closest distance between any charger and the closest collector to the trash, T_e is the elapsed flight time since take off, s is the UAV speed, and C_1 a constant factor added to account for any uncertainty in these parameters or numerical limitations. Since these terms are all known in this scenario, C_1 was set to one to account for any numerical computational errors. The assumption that the distances can be calculated accurately stems from the assumption that a UAV has an internal map of the park and a good estimate of where the trash is from its sensors.

$$T_F - T_e \geq \frac{d_{UAV,t} + \min(d_{t,c}) + \min(d_{c,r})}{s} + \tau_{rt} + \tau_{dt} + \tau_c + C_1 \tag{2.1}$$

If Eq. 2.1 is satisfied, the UAV sets the trash target as its goal and flies towards it. If a UAV detects closer trash on its way to the target, it evaluates Eq. 2.1 again with the position of the closer trash, and if the inequality is satisfied, the UAV updates its goal to this closer trash target. Once the UAV reaches the trash, it retrieves it during τ_{rt}. The UAV then travels to the closest collector and deposits the trash in the collector on arrival, after which it sets out again to search according to its specified searching strategy.

While the UAV is searching, it evaluates Eq. 2.2 at each time step, where $min(d_{UAV,r})$ is the distance between the UAV and the closest charging station, converted to time of flight to the station by dividing it by the nominal UAV speed, τ_c is the time it takes to land on the charging station and C_2 is a safety constant, similar to C_1. C_2 was also set to one second for this case study. If Eq. 2.2 is true, the UAV returns to the closest charging station. After traveling to and landing on the charging pad, the UAV proceeds to charge until full and then returns to search.

$$T_F - T_e < \frac{\min(d_{UAV,r})}{s} + \tau_c + C_2 \tag{2.2}$$

If all the UAVs were deployed at the same time, after searching for T_F their energy would be depleted at the same time and they would all need to recharge simultaneously. During the period of recharging there would be no UAVs to search the park and retrieve targets. To avoid this situation, the UAVs are split up into deployment groups that start searching at staggered times. This guarantees that at least some UAVs will be deployed at all times. The number of groups is dependent on the ratio of the recharge time to the fight time and with $T_r = 60$ and $T_F = 30$, the ratio is 2. This means that two groups of UAVs are required to search the area in the time it takes one group of UAVs to recharge. Therefore, a minimum of three groups of UAVs are needed in total to have UAVs continually deployed.

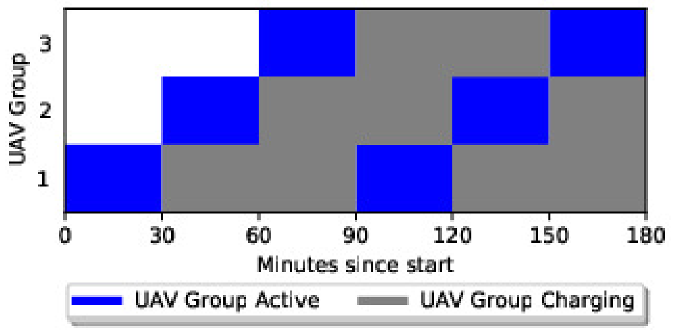
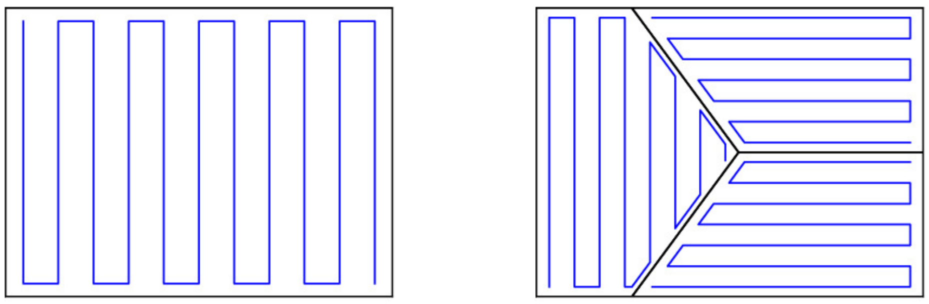


Figure 2.3: UAV deployment schedule for three UAV groups



(a) Global lawnmower

(b) Partitioned lawnmower

Figure 2.4: Search patterns

Given one UAV group has full energy, one group is charging with half energy, and the other has just returned from searching, the UAV group with full energy can search until the its group's energies are depleted. After this, the group with half energy will have full energy and can take the place of the group with no energy, and the cycle can repeat. A visualization of this scheduling process is shown in Fig. 2.3.

2.5.2 UAV Search Strategies

As part of this case study, four search strategies were implemented and evaluated. The first, called random bounce, consists of each UAV proceeding on a straight line until it reaches the edge of the environment, then choosing a random angle facing towards a different edge of the environment and heading in a straight line in that direction. This is repeated for as long as the UAV is searching. In the second search strategy, called global lawnmower, the UAVs follow a lawnmower path through the entire park as shown in Fig. 2.4a. They are initialized to start their searches on the path with equal distances between them as measured on the path length of the lawnmower pattern to spread evenly out.

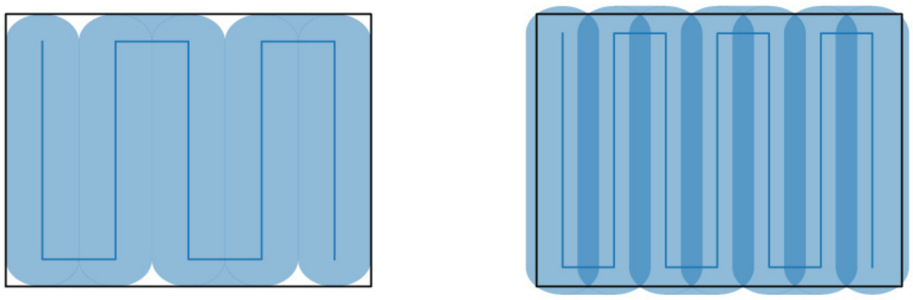
With the third search algorithm, called partitioned lawnmower, the area is partitioned into subdivisions. The number of subdivisions is equal to the number of UAVs patrolling as seen in Fig. 2.4b. Finally, in the fourth strategy, named partitioned bounce, the area is likewise partitioned into subdivisions but the UAVs follow the strategy of random bounce within their partitions. The partitions were created through a Voronoi diagram with Voronoi vertices being the points chosen with the algorithm explained in section 2.5.3.

The algorithm to generate the lawnmower pattern for global lawnmower and partitioned lawnmower was based on an algorithm (labeled "algorithm A" in the referenced paper) by Di et al. [85] and modified to be dependent on r_d. The algorithm was designed to function in convex polygons since for global lawnmower, the lawnmower pattern is in a square, and for partitioned lawnmower, the partitions are always convex due to Voronoi regions always being convex [86]. One major change to the original algorithm is that if the distances from all the vertices to the midpoint were less than $2r_d$, a spiral pattern was used since the lawnmower algorithm had a high probability of not covering the whole area in these situations. The algorithm is referenced in Algorithm 1, where $d(c, v)$ is the distance between the centroid and a vertex, $d_{\perp_{max}}$ is the longest perpendicular distance between the longest edge and any vertex, d_{edge} is the distance from the edges to the search pattern, and d_{lane} is the distance in between each long pass over the area.

Galceran et al. mention critical points that are not covered in the lawnmower pattern if d_{edge} is r_d and $d_{lane} = 2r_d$ in a square [87]. This can be fixed by dividing d_{edge} and d_{lane} by $\sqrt{2}$, which guarantees this distance is always covered in a square, at the cost of adding extra lanes. Fig. 2.5 illustrates this change. If the convex polygon is not a square shape, critical points could be inclined on a slope such as in the patterns in Fig. 2.4b. In this situation, even with a multiplicative correction term of $\frac{1}{\sqrt{2}}$ applied to d_{edge} and d_{lane}, there will still be uncovered critical points. In this case an extra term C_3, that can contain a value such that $0 < C_3 \leq 1$, was multiplied to d_{edge} to adjust it so that the whole space is covered.

Algorithm 1: Search Pattern Generation for a Convex Polygon

```
Input: $r_d$, set of vertices $\vec{V}$ for a simple convex polygon, centroid of polygon $c$
Output: Ordered list of waypoints for search pattern
if $d(c, v) < (2r_d), \forall v \in \vec{V}$ then
    Construct Spiral Pattern;
    for $v \in \vec{V}$ do
        if $d(c, v) < r_d$ then
            insert $c$ into point set if not already exists;
        else
            direction $\leftarrow \frac{c-v}{d(c,v)}$;
            point to add $\leftarrow (v + r_d *$direction$)$;
            insert point into point set;
        end
    end
else
    Construct Lawnmower Pattern;
    $e_{max} \leftarrow$ Longest edge;
    $d_{\perp_{max}} \leftarrow \max(d_{\perp}(e_{max}, v), \forall v \in \vec{V})$ ;
    $d_{edge} = C_3 * \frac{r_d}{\sqrt{(2)}}$;
    $d_{lane} = \frac{2*r_d}{\sqrt{(2)}}$;
    $N_{lanes} = 1 + Round(\frac{(d_{\perp_{max}} - 2*d_{edge})}{d_{lane}})$;
    $\vec{t} \leftarrow$ tangent direction of $e_{max}$ facing inside polygon;
    $V_{curr} \leftarrow$ vertices of $e_{max}$;
    for $i \leftarrow 1$ to $N_{lanes}$ by $1$ do
        if $i = 1$ then
            $V_{curr} += d_{edge} * \vec{t}$
        else
            $V_{curr} += d_{lane} * \vec{t}$
        end
        $l_V \leftarrow$ line connecting $V_{curr}$;
        $l_I \leftarrow$ line formed from intersection points with polygon when $l_V$ is extended infinitely;
        $m \leftarrow$ midpoint of $l_I$;
        if $length(l_I) > 2 * d_{edge}$ then
            $P_{toAdd} \leftarrow$ endpoints of $l_I$ translated towards $m$ by $d_{edge}$;
            if $i \bmod 2 = 0$ then
                Add $P_{toAdd_1}$ then $P_{toAdd_2}$ to point set;
            else
                Add $P_{toAdd_2}$ then $P_{toAdd_1}$ to point set;
            end
        else
            Add $m$ to point set;
        end
    end
end
return point set;
```



(a) Original pattern with critical areas uncovered (i.e. not searched by a UAV)

(b) Modified lane width with critical areas covered (with the cost of overlap)

Figure 2.5: Lawnmower comparison before and after modified lane width. Outside border represents the park boundaries

Adjusting d_{edge} is advantageous rather than adjusting d_{lane}, so that extra lanes will not have to be added to accommodate the extra critical points. C_3 was experimentally set at 0.6 to make sure the area was covered across all the full range of park sizes and UAVs search radius values. One drawback to this approach is that there is a small possibility points might not be covered, but in all cases that were evaluated the area was negligible.

2.5.3 Collector and Charger Placement Algorithm

During the course of the simulation, UAVs will travel to collectors and chargers many times and therefore it is important to optimally place them so that less time will be spent depositing targets and travelling to charging stations and more time spent searching for targets. Consider a square space with a set of positions, $\mathcal{M}$, containing a number of positions N_p, each defined in $\mathbb{R}^2$ within the park. Since a UAV flies to the closest collector after picking up trash, and flies to the closest charging station with low energy, the charger and collector positions should be placed in a way than minimizes the average distance from the locations where these events will likely occur to their closest positions in $\mathcal{M}$.

The average distance from any point to its closest position of interest in $\mathcal{M}$ weighted by the probability of events occurring in certain locations, d_{avg}, can be defined as an integral $d_{avg} = \frac{1}{A_\mathcal{P}} \int_0^{l_\mathcal{P}} \int_0^{l_\mathcal{P}} w(x, y) \min(d((x, y), \mathcal{M}))\mathrm{d}x\mathrm{d}y$, where $A_\mathcal{P}$ is the area of the park,

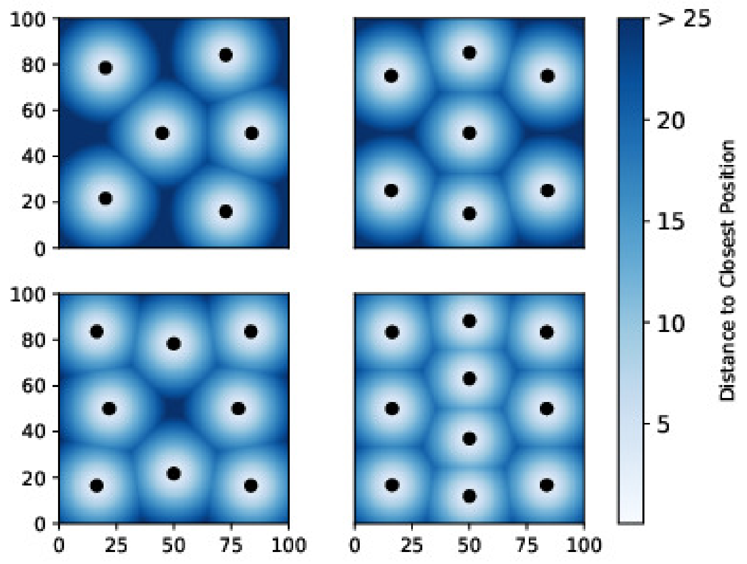


Figure 2.6: Optimized positions with heat map of the distance from each grid cell to the closest position in $\mathcal{M}$

$\min(d((x, y), \mathcal{M})$ is the Euclidean distance from a point defined by coordinates (x, y) to the closest position in $\mathcal{M}$, and w is a weighting function the depends on the probability of the event happening at that location.

Since targets appear with a uniform random distribution, and appear according to a binomial distribution with an expected value of γ trash per hour, $w(x, y)$ is constant and can be pulled out of this integral. The probability of a UAV deciding to charge for a certain location can be complicated to model since it is dependent on the UAV search path and the stochastic nature of the target appearance model, but a conservative estimate is to treat the whole area with equal probability as with the target appearance model. This assumption will be made for the purposes of this case study, and so d_{avg} can be considered equivalent for charger and collector placement. This integral can be evaluated discretely by dividing up the area into grid cells and calculating the distance from each grid cell to its closest position of importance, and taking the average of these distances. If the spatial probability of an event occurring were to be different than a uniform distribution, $w(x, y)$ could be discretely

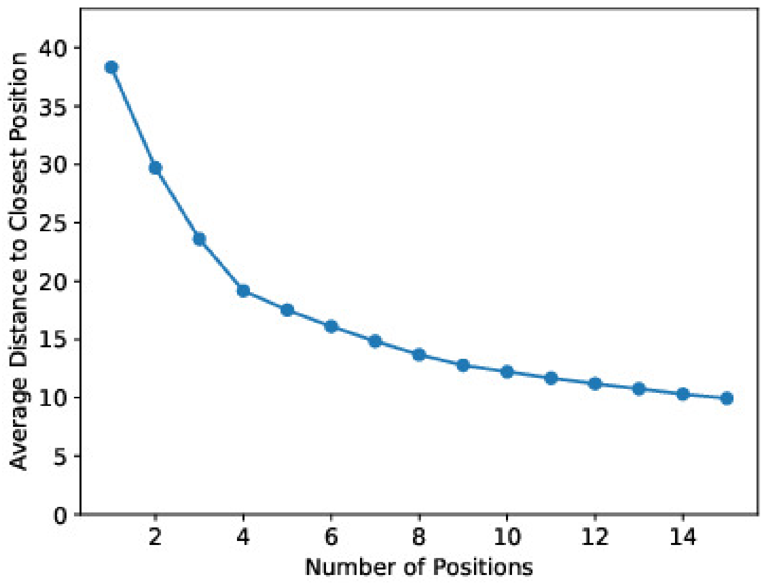


Figure 2.7: Objective function minimized with different numbers of positions

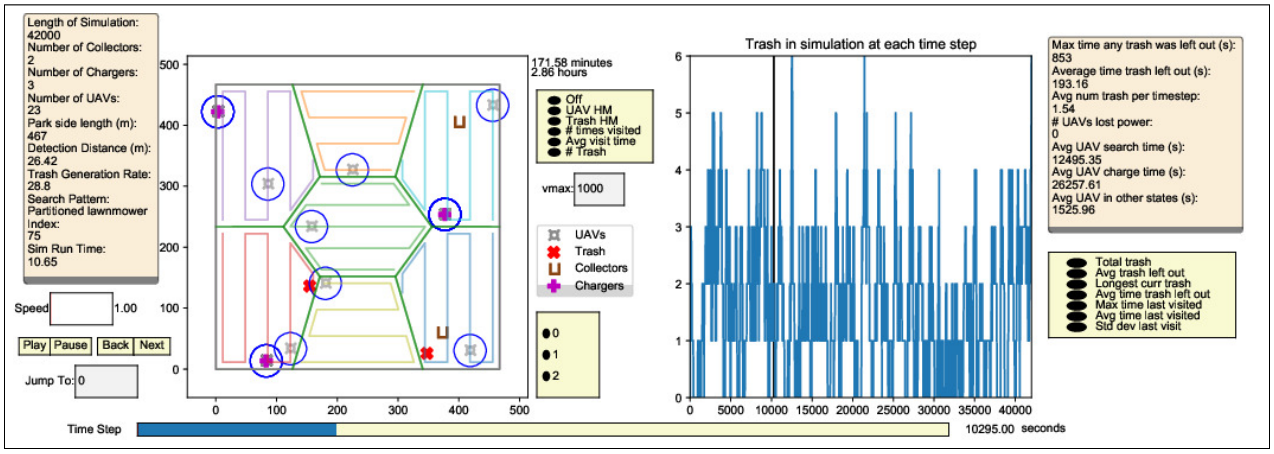
approximated in each grid cell and multiplied with the distance of the grid cell to its closest position of importance.

An optimization problem was formulated to choose $\mathcal{M}$ to minimize the objective function. This is shown in Eq. 2.3 where the area is divided into a set of square grid cells $\mathcal{G}$ of equal area, with $\min(d(g, \mathcal{M}))$ being the minimum Euclidean distance from the centroid of a grid cell to any charger or collector position in $\mathcal{M}$.

$$\begin{aligned} \underset{\mathcal{M}}{\text{minimize}} \quad & f(\mathcal{M}) = \frac{1}{|\mathcal{G}|} \sum_{g \in \mathcal{G}} \min(d(g, \mathcal{M})) \\ \text{subject to} \quad & 0 \leq m_x \leq l_{\mathcal{P}}, \quad \forall m \in \mathcal{M}, \\ & 0 \leq m_y \leq l_{\mathcal{P}}, \quad \forall m \in \mathcal{M} \end{aligned} \tag{2.3}$$

A two-step optimization was performed to minimize this objective function. First, an initial solution was found with a differential evolution algorithm from scipy's optimize package [88]. In this first step, $|\mathcal{G}|$ was defined as 900 grid cells, equivalent to a 30 by 30 grid, to reduce computation time for the initial approximate solution. After the approximate solution was found a convex minimization algorithm, the SLSQP method from scipy's optimize package [89], with a much finer grid discretization was used with the initial approximate solution as a starting point to find the local minimum in that area.

Examples of resulting position placement from optimizing the objective function are shown in Fig. 2.6. The objective function values from this optimization with increasing N_p are shown in Fig. 2.7 for a park with $l_{\mathcal{P}} = 100$ m. This generally follows an exponential slope downwards, with larger decreases seen when N_p is closer to zero, and smaller decreases with increasing N_p. However, if the area of the park $A_{\mathcal{P}}$ is large, depending on the cost, adding an extra position even with a high nominal N_p may be worth the decrease in average distance.


Length of Simulation:
42000
Number of Collectors:
2
Number of Chargers:
3
Number of UAVs:
23
Park side length (m):
467
Detection Distance (m):
26.42
Trash Generation Rate:
28.8
Search Pattern:
Partitioned lawnmower
Index:
75
Sim Run Time:
10.65

Speed 1.00

Jump To: 0

171.58 minutes
2.86 hours

vmax: 1000

Max time any trash was left out (s):
853
Average time trash left out (s):
193.16
Avg num trash per timestep:
1.54
UAVs lost power:
0
Avg UAV search time (s):
12495.35
Avg UAV charge time (s):
26257.61
Avg UAV in other states (s):
1525.96

Time Step 10295.00 seconds

Figure 2.8: Screenshot of interactive GUI

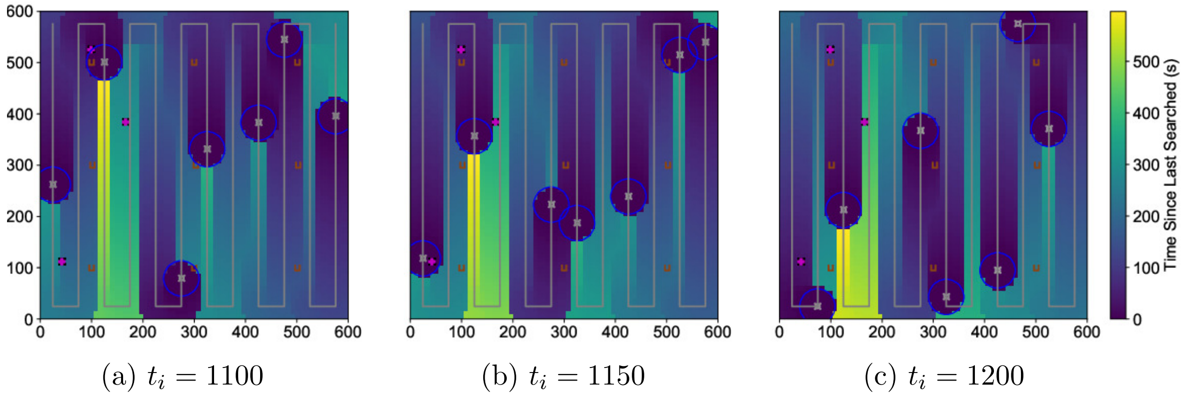
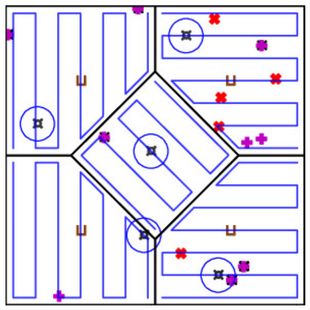
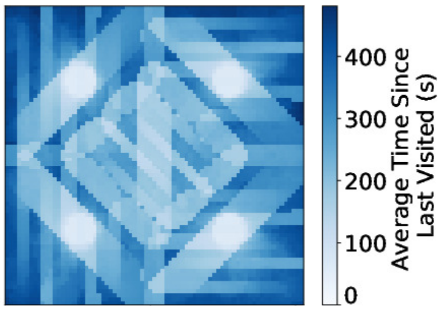


Figure 2.9: Heat maps over time where the value of each grid cell is the time since the cell was last searched by a UAV


(a) Partitioned lawnmower search pattern



(b) Average time since last searched heat map

Figure 2.10: Results for simulation with 15 UAVs and $\gamma = 12.36$, note that the the locations where collectors are present are searched more often since the UAVs search after they deposit trash in the collector

2.5.4 User Interface and Simulation Exploration

A graphical user interface (GUI) was created for visualization of simulation behavior, along with charts for exploratory analysis. A screen shot of the GUI is shown in Fig. 2.8. The left section of the GUI shows the park, a 2D square, with UAV agents symbolized by the gray four pointed symbol. The circle around each UAV represents the boundary of its detection area dependent on r_d. The search patterns of each UAV group are plotted; in Fig. 2.8 the partitioned lawnmower patrol paths are displayed. The other elements positions are as shown in the left section, and represented by the legend in the middle. The right side of the GUI has an adjustable line chart that displays how a specified value changes over time. In Fig. 2.8, the chart is set as the number of trash in the simulation at each time step, which can be examined to quickly know at what time steps the number of trash in the simulation was high. The slider bar and buttons can then be used to navigate to those time steps and understand the patterns or behaviors that caused the high values.

Optional heat maps can be toggled on and off in the left section of the GUI, used to visualize spatial information. One heat map displays data dependent on when the UAV last searched a grid cell from a set of equal area grid cells $\mathcal{G}$ in $\mathcal{P}$. Whenever a grid cell in $\mathcal{G}$ was less than a number of cells away from the UAV while it was searching, meeting the equality described to Eq. 2.6, it was reset to zero, and all other cells add one to their value at each time step. In Eq. 2.6, r_g is the detection distance converted to number of grid cells, x_g is the number of horizontal grid cells from the grid cell containing the UAV, and y_g is the number of horizontal grid cells from the grid cell containing the UAV. The grid cell radius, r_g is resultant from Eq. 2.5, where l_g is the length of a grid cell, or the ratio of the the park length, $l_{\mathcal{P}}$ and the number of cells in a row of cells, $\sqrt{|\mathcal{G}|}$. $\frac{r_d}{l_g}$ was rounded since $\frac{r_d}{l_g}$ is usually not an integer, and one was added to make sure that the UAV wouldn't miss grid cells that were actually searched.

$$l_g = \frac{l_{\mathcal{P}}}{\sqrt{|\mathcal{G}|}} \tag{2.4}$$

$$r_g = Round(\frac{r_d}{l_g}) + 1 \tag{2.5}$$

$$\sqrt{(x_g^2 + y_g^2)} \leq r_g \tag{2.6}$$

Three selected times from the full time series of these heat maps are shown in Fig. 2.9. From these heat map visualizations, the areas of the park that have not been searched for a long period of time can be identified by the lighter hues. The time history for this heat map is recorded for each grid cell, which enables a heat map display of the average last search time for each cell over the entire simulation, providing a high-level output metric of how the UAVs performed overall. An example of such a heat map is shown in Fig. 2.10b with the associated scenario separated into Fig. 2.10a for clarity. In Fig. 2.10b the locations where the UAVs crossed their partition from the end to the beginning of their lawnmower patterns have a lower average search time because these segments overlap areas already searched in the lawnmower pattern. Four lighter spots can also be seen around the collector positions, since UAVs start searching for trash immediately upon depositing trash into the collectors, and γ was high enough, in this example, so that there were frequent visits to each collector. It can also be seen that the overall partition in the center is lighter, which suggests it takes less time for the UAV in that partition to cover its space, on average, and correlates to a smaller area as compared to the outer partitions. These hue differences identify areas for improvement in the lawnmower and partitioning algorithms, since ideally there would be no overlap with the lawnmower pattern and the partitions would be equal area.

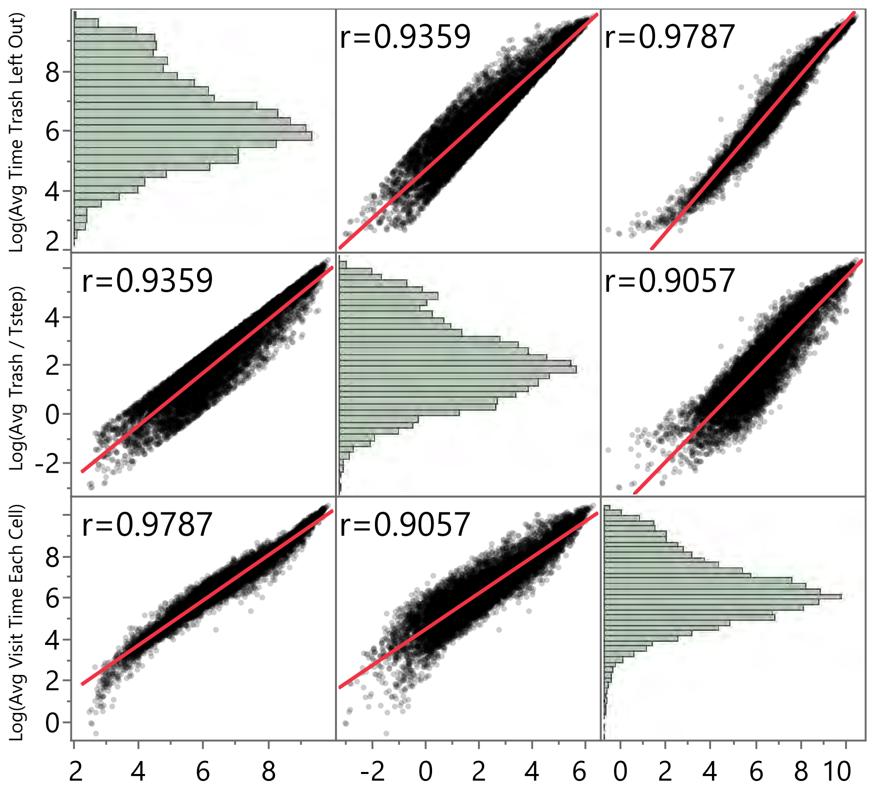


Figure 2.11: Correlations of log(outputs)

Table 2.1: Continuous parameters with upper and lower limits for LHS DOE

Parameter	**Lower Limit**	**Upper Limit**	**Unit**
N_C	1	10	Collectors
N_R	1	10	Chargers
N_{UAV}	3	27	UAVs
$l_{\mathcal{P}}$	200	800	Meters
γ	10.8	108.0	Trash/Hour
r_d	10	50	Meters

Table 2.2: Discrete parameters with associated levels for LHS DOE

Parameter	**Setting**
Search Pattern	- Random Bounce
	- Global Lawnmower
	- Partitioned Bounce
	- Partitioned Lawnmower
Charger Placement	- Optimized
	- Random
Collector Placement	- Optimized
	- Random

2.5.5 System Analysis and Verification

A design of experiments was created and executed to understand the impact of search pattern and other parameters on effectiveness. The latin hypercube sampling (LHS) technique, which uniformly samples the design space [90], was chosen to generate parameter values for each simulation. Nine parameters were chosen as variable inputs to the simulation, shown in Tab. 2.1 and Tab. 2.2. The simulations were run for $T_S = 42000$ seconds, corresponding to about one business day of operation for a park, 11.66 hours, with a time step of one second. 5000 experiments, repeated twice, each with different random seeds which caused trash to appear at the same rate but in different places, were performed for a total of 10000 simulations.

A number of aggregate outputs measured in each simulation were chosen to quantify effectiveness. The first of these measures relates to the set of trash $\mathcal{Q}$ that appeared in the simulation over all time steps as the average time of trash retrieval, $\overline{T}_r$, defined as $\overline{T}_r = \frac{1}{|\mathcal{Q}|}\sum_{t\in\mathcal{Q}} T_r^t$, where T_r^t is the amount of time from the appearance of trash t to its retrieval by a UAV. The second effectiveness metric explored, also related to $\mathcal{Q}$, was the average number of trash left out at each time step, $\overline{N}_t$, defined as $\overline{N}_t = \frac{1}{T_S}\sum_{i=1}^{T_S} |\mathcal{Q}_i|$, where $\mathcal{Q}_i$ is the set of trash left out at time step i. The third metric chosen was the average time any area in the simulation was last searched, $\overline{T}_v$. This is defined in $\overline{T}_v = \frac{1}{|\mathcal{G}|T_S}\sum_{g\in\mathcal{G}}\sum_{i=0}^{T_S} T_v^{g,i}$, where $\mathcal{G}$ is the set of discretized grid cells, similar to Eq. 2.3, and $T_v^{g,i}$ is the amount of time since cell g had been searched last by a UAV at time step i. The value of the cell is reset to zero time (since last searched) with the same methodology introduced for calculating the heat maps in Fig. 2.9.

Examining the correlation of the outputs over all simulation performed in Fig. 2.11 revealed that the log of the outputs were all highly correlated with r-values higher than 0.9 and p-values of less than 0.0001. This shows that the outputs under question are highly related. Multiple linear regression was applied to the $\log(\overline{T}_r)$ using JMP, a statistical program, to understand how the input variables affected this output and to validate model assumptions. The parameters chosen for the regression model were the first order effects included in the DOE, and the $(N_{UAV})^2$ second-order effect. The R-squared value calculated from the fit of this model was 0.92. A rich model of all parameters and their second order

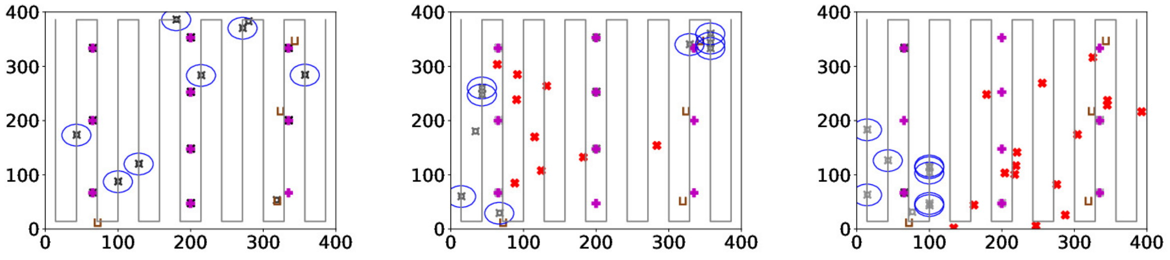
Table 2.3: Regression results for $\overline{T}_r$ with confidence intervals (CI)

Term	**Estimate**	**p-Value**	**Lower 95% CI**	**Upper 95% CI**
Intercept	2,387.485	<.0001	2,253.526	2,529.407
Optimized Collector Placement	0.813	<.0001	0.800	0.826
Optimized Charger Placement	0.981	0.0231	0.966	0.997
N_R	0.996	0.0123	0.994	0.999
N_C	0.934	<.0001	0.931	0.936
N_{UAV}	0.752	<.0001	0.748	0.756
$(N_{UAV})^2$	1.005	<.0001	1.005	1.006
r_d	0.969	<.0001	0.968	0.969
$l_{\mathcal{P}}$	1.005	<.0001	1.005	1.005
γ	1.011	<.0001	1.011	1.012
Search Pattern[Partitioned Bounce]	0.758	<.0001	0.741	0.776
Search Pattern[Partitioned Lawnmower]	0.600	<.0001	0.586	0.614
Search Pattern[Random Bounce]	0.723	<.0001	0.707	0.740

Table 2.4: Search pattern comparison with Tukey HSD test with Confidence Intervals (CI)

Level	**Comparison Level**	**Est. Ratio**	**Lower 95% CI**	**Upper 95% CI**	**p-Value**
Global Lawnmower	Partitioned Lawnmower	1.667	1.618	1.718	<.0001
Global Lawnmower	Random Bounce	1.383	1.343	1.425	<.0001
Global Lawnmower	Partitioned Bounce	1.319	1.280	1.359	<.0001
Partitioned Bounce	Partitioned Lawnmower	1.264	1.227	1.303	<.0001
Random Bounce	Partitioned Lawnmower	1.205	1.170	1.242	<.0001
Partitioned Bounce	Random Bounce	1.049	1.018	1.081	0.0002

effects was fitted, but it only increased the R-squared value of the original fit by 0.03, and so the simplified model was deemed sufficient and kept for subsequent analysis. In this fit there is a strong correlation between each parameter and $\log(\overline{T}_r)$, implying that for each unit increase in a parameter there is a multiplicative increase in $\overline{T}_r$ with a magnitude unique to each parameter and expressed by the estimates in Tab. 2.3. γ, $l_{\mathcal{P}}$, and r_d all had a significant practical effect on $\overline{T}_r$. This helped to verify the model, since these variables have strong intuitive correlations with effectiveness. Bigger parks from increased $l_{\mathcal{P}}$ require more time for UAVs to search, higher γ causes UAVs to spend more time retrieving trash targets, which leave less time to search, and smaller values of r_d lead to longer travel distances and more time to search a full park or a partition. Increased trash retrieval times and travel distances increase $\overline{T}_r$, which is reflected with multiplicative effects on $\overline{T}_r$ greater than 1.0 with γ and $l_{\mathcal{P}}$, and less than 1.0 with r_d for unit increases in those parameters.



(a) $t_i = 300$ (b) $t_i = 20300$ (c) $t_i = 40300$

Figure 2.12: Global lawnmower search pattern stack up effect over time

Although the linear regression results describe the comparative effects of each search pattern on $\overline{T}_r$ compared to the reference level, global lawnmower, Tukey's honestly significant difference (HSD) test was performed to adjust p-values and confidence intervals for multiple comparisons [91]. The results of this test are shown in Tab. 2.4. According to the results it is highly suggestive that random bounce, partitioned bounce, and partitioned lawnmower patterns had a larger reductive effect on $\overline{T}_r$ compared to the global lawnmower search pattern. Tukey's HSD test also strongly suggests that the partitioned lawnmower has a larger reductive effect on $\overline{T}_r$ compared to random bounce and partitioned bounce, and that there is a small but statistically significant difference between how random bounce and partitioned bounce affected $\overline{T}_r$.

Examining global lawnmower more closely revealed why it performed much worse than the other patterns. When a UAV detects trash as it traverses the global lawnmower pattern, it retrieves and deposits it, and then returns to the same place on the global lawnmower pattern that it started on when it detected the trash. During that time of retrieval and deposit, UAVs following along the same path will decrease the distance gap between them so that when the first UAV returns, the UAVs will be much closer to the first as they continue the search. Over time, this behavior causes the UAVs to stack on top of each other as seen in Fig. 2.12 and effectively reduces the percentage of the park that is searched at each time step. With higher γ, this was even more pronounced. To avoid the stacking phenomenon, an optimized strategy would need to be developed for the global lawnmower search pattern that intelligently decides where UAVs should return to search after retrieving a target. A first order strategy could include a return to the projected point further down the path had the UAV not detected any trash.

2.6 Discussion and Future Iterations

One interesting result from the system analysis was that the number of charging stations and whether they were placed randomly had a small influence on UAV effectiveness. N_R had a small reductive effect of 0.996 on $\overline{T}_r$ for each charger added, and the estimate of the effect between optimized and non-optimized charger placement on $\overline{T}_r$ was 0.9815. This could be attributed to the size of the parks being studied. In every scenario examined, each UAV was able to patrol their area multiple times before having to charge. Since the UAVs only travel between their search areas and charging stations twice every 30 minutes, it follows that the distance to any individual charger would not have a large influence on effectiveness. This could have been a bigger factor if a significant portion of the flight time was used to fly to and from charging stations and partitions due to large park sizes and short T_F.

The number and placement of the collector stations, however, made a significant impact on effectiveness metrics. It is strongly suggestive that optimized collector locations helped lower $\overline{T}_r$, with a p-value of less than 0.0001 and a multiplicative effect of 0.934 on $\overline{T}_r$ for each collector added. Placing collectors with the optimized locations also had a 0.813 multiplicative effect on $\overline{T}_r$ compared to a random collector placement. Along with the collectors, each additional UAV had a 0.752 multiplicative reduction in $\overline{T}_r$, which was a large practical difference compared to other parameters. The second-order effect of $(N_{UAV})^2$ with an estimate of 1.005 shows that the benefits of adding a UAV slightly decrease as more UAVs are added to the scenario, but overall there were large benefits for each UAV added. These observations lead to the conclusion that if resources are constrained for charger, collector, and UAV acquisition in park trash retrieval, resources should be put first to UAVs, then to collectors, and chargers last, and that chargers and collectors should always be placed according to the optimized methodology discussed in Sect. 2.5.3 as opposed to randomly.

Many elements of the simulation design framework (refer to Fig. 2.1) not included in the experiment would likewise influence effectiveness. More constraints on the maneuver models and risk of failure during the maneuvers would mean less efficient searching and less time to find targets. Avoiding obstacles such as humans, animals, or trees would also increase search time. More realistic object detection models that involve probabilistic detection would make planning search patterns more difficult since it is not guaranteed to find a target in a searched area. A real-time optimization algorithm could be more effective than the deterministic search patterns presented since a real-time algorithm makes decisions about where to search based on global information, rather than following a pre-planned pattern. However, this method would increase computational costs and is left for future studies. Multi-UAV interactions including sharing of information during return and drop-off segments could have increased effectiveness of the system. Thus, if UAVs had memory of previous trash seen and could communicate this to other UAVs, this could greatly increase the effectiveness of the system, assuming the communication is reliable. If this knowledge were incorporated in the searching strategy, this could cause even greater improvements. In the future, these elements should be considered and the cost-benefit trade-off of each feature examined for PSR-TSA.

2.7 Conclusion

In this chapter a framework for exploring the multi-UAV persistent search and retrieval task with stochastic target appearance was presented and discussed. The use of graphical and statistical analysis techniques were demonstrated to verify and evaluate system effectiveness. A case study was executed, with comparison testing of four search patterns within the constraints of the framework. Statistical methods showed the partitioned lawnmower search pattern performed the best compared to other search patterns, and the influence of various parameters on overall effectiveness metrics suggested that increasing the number of UAVs is, initially, the best investment strategy over increasing charger or collector locations for typical park sizes.

CHAPTER 3. SPATIOTEMPORAL ANALYSIS OF MULTI-UAV PERSISTENT SEARCH AND RETRIEVAL WITH STOCHASTIC TARGET APPEARANCE

3.1 Preface

The probabilistic nature of multi-UAV PSR-STA task introduces non-deterministic elements in the multi-UAV search behavior that can make it difficult to analyze. Measures that summarize the effectiveness of a multi-UAV PSR-STA scenario with one value can be useful for an initial analysis, but may not be enough to fully understand the situation since these measures do not adequately capture the variations of effectiveness over the area and time period of the scenario. This chapter analyzes multi-UAV PSR-STA with methods based on dimensionality reduction techniques and graphical comparison that are capable of analyzing temporal and spatial trends in multi-UAV search effectiveness across a range of scenarios. For temporal analysis, line charts are used for graphical comparison of temporal patterns over a range of scenarios, and the discrete Fourier transform is used to identify shared temporal signals. For spatial analysis, principal component analysis and a random forest surrogate model with a profiler is used to explore the non-linear influence of input parameters on spatial patterns. A trellis chart or figure of figures is used for graphical comparison of both temporal and spatial patterns. Temporal and spatial measures tailored for multi-UAV PSR-STA are introduced that enable these analysis techniques. This chapter builds on the methods developed in chapter 2.

3.2 Introduction

Groups of small, autonomous, battery powered unmanned air vehicles (UAVs) are increasingly used in many application areas [23, 92]. One such area is the persistent search and retrieval task with stochastic target appearance (PSR-STA) [93]. In this scenario, UAVs search an area for stochastically appearing targets of interest to retrieve and deliver these targets to a collector location. An example of an application that motivates the study of multi-UAV PSR-STA is litter removal, where litter is dislodged by wind or discarded by people in an area [8] and retrieved and deposited into a trash bin by a UAV or other autonomous agent [9]. A study prepared for the Environmental Protection Agency estimated that west coast communities in the United States of America spend more than \$520,000,000 each year to combat littering, and hundreds of species of animals are affected as the litter is eventually displaced to the ocean [11]. This emphasizes the need for studying and understanding multi-UAV PSR-STA for successful deployment of UAVs to help with this task, as UAVs relative low cost and ability to interact with the environment without an operator would help to improve communities and reduce cost through autonomous litter collection.

Since testing many variations of multi-UAV search scenarios in the real world is time and cost prohibitive, a common methodology for understanding the effectiveness of a UAV search task is to create a computer simulation of the problem domain and run the simulation many times according to a Monte Carlo approach or other simulation exploration technique, varying chosen parameters while recording outputs of interest in each simulation [16–20]. Potential causal and corollary relationships can then be established among the inputs and the outputs, and trends can be understood about which inputs are most influential to the responses. From these analyses, conclusions can be made about which parameters have the largest impact on effectiveness over a range of scenarios. This approach is an efficient way to compare search algorithms, providing understanding into how parameters influence overall search effectiveness and enabling many other insights into search algorithm performance. However, if these patterns are to be implemented in real world scenarios, detailed analyses that reveal information about spatial and temporal variations and patterns inherent in the search behavior beyond simple quantification of effectiveness are desirable.

When search patterns follow a deterministic path, spatial and temporal pattern analysis is not as important since metrics of effectiveness are easily defined and UAV behavior is deterministic. With multi-UAV PSR-STA, non-deterministic search behavior is present even with deterministic coverage search patterns since UAVs must pause their search for a significant amount of time when retrieving targets and delivering them to a collector location.

Table 3.1: Summary of analysis methods used in this research

Type of Analysis	**Methods Used**
Temporal analysis	-Discrete Fourier transform -Line chart examination
Spatial analysis	-Principal component analysis -Random forest surrogate model with a profiler -Heat map examination
Temporal and spatial analysis	-Trellis charts (figure of figures)

Because of the delays in searching due to retrieving and delivering targets, the multi-UAV search behavior does not follow an easily understood deterministic pattern, which motivates the need to understand spatiotemporal variations in effectiveness in multi-UAV PSR-STA. The location and number of resources such as collectors [93] and chargers [14] can also influence search effectiveness, which further complicates analysis. Some research has compared time or spatial trends dependent on UAV search algorithms [49] and target appearance models [55] for individual simulations. This work extends the exploration of spatiotemporal trends for individual simulations in identifying and comparing trends over a wide range of scenarios and parameters for multi-UAV PSR-STA.

It can be difficult to (1) identify spatial and temporal patterns resultant from UAV search and (2) attribute the influence of varied input parameters to these patterns since spatial and temporal patterns exist in high-dimensional spaces. This research aims to identify and analyze patterns existent in multi-UAV PSR-STA over time and space by characterizing high-dimensional spatiotemporal data in understandable and comparable lower dimensions, extending metrics developed in [93] for spatiotemporal analysis, and presenting graphical techniques to compare trends common among many scenarios. A summary of the analysis methods used in this research is given in Tab. 3.1. Further introduction and explanation of each method are given in sections 3.5 and 3.6.

This work builds on previous research of a framework and basic analysis methods for multi-UAV PSR-STA by Day and Salmon [93]. It applies the problem specification and algorithms from the previous work and reintroduces the metrics of effectiveness established in the previous chapter, broadening their scope for use in identifying spatiotemporal trends.

3.3 Related Works

Research related to UAV search uses various metrics of effectiveness and analysis methods to understand the behavior of the search algorithms. One metric discussed is refresh time [94], also known as the time since an area was last visited. To measure this metric, the area is discretized into square grid cells, and at certain intervals in the simulation, the time since each grid cell was last visited by a UAV is recorded [49]. The criteria for when a UAV has visited or searched a grid cell can be difficult to define, since a UAV's detection area is not always aligned with the arbitrary grid structure imposed for measurement. Some techniques only count the grid cells as visited when the cell is completely covered by the UAV's detection area [95]. Others define the cells to be the same size as the detection area of the UAV, and similarly only are counted as visited when the UAV detection area fully overlaps the specific grid cell [6]. Waharte et al. proposed measures that account for when a UAV's search area is mostly in one cell, but overlaps other cells [96], but admitted that their strategy was inferior to the best strategy, which was to introduce new grid cells that matched the grid structure of overlaps at each time step. This best strategy was determined to be computationally infeasible.

A related metric to refresh time is to have the value of each grid cell set at a constant non-zero value if they are covered by the UAV search area and have the other uncovered cell values decay linearly at each time step according to a constant, as was applied by Gainer et al. [46], to examine relationships between coverage and UAV operation. Another metric of effectiveness is to record the maximum value of the refresh time of any cell at each time step, with specific subsections of interest having their own maximum refresh time, which can be plotted to understand the oscillatory nature of persistent UAV search [97]. When the search is probabilistic, the metric of information gained or the probability of detection [54] can be considered, as well as a measure called awareness that is related to information entropy [98]. There are also many domain specific related measures of effectiveness such as the size of burnt land for a forest fighting mission [16], the number of targets tracked over time for a search and track task [46], and the average delay when a stochastically appearing target appears and when it is observed in a mobile sensing task [55].

As mentioned in Sec. 3.2, a common way to analyze a scenario is to perform many simulations, varying the parameters of interest, and then analyzing the resultant data in bulk from the scenarios [18, 51]. If domain specific measures of effectiveness exist, these can be examined to understand which parameters are most influential on effectiveness. One way to examine the input parameters is to plot the output of interest in relation to an input parameter, with box plots or confidence intervals showing the range of outputs from multiple simulations for the input [22]. Multiple line plots could also be simultaneously plotted for different levels of a parameter of interest [99]. This is useful when the number of input parameters are sufficiently low, but patterns can remain overlooked if these plots are the only methods used to visualize higher dimensional data. To examine an output variable that is affected non-linearly, by multiple input variables, a common strategy is to fit a surrogate model to this output with the different simulation parameters as the inputs, and then exercise a profiler tool to understand how the inputs affect the outputs [16]. This profiler displays the non-linear effects on the output from the reference level. It can be dynamically explored to understand how trends change depending on differing parameter values in the design space. Another similar method is to compare effect plots, which are profilers but shown at certain levels [17]. While these are not tools for summarizing the entire design space, they are effective for understanding non-linear trends and to identify areas for further exploration.

Heat map comparison can likewise be used for comparing simulations, which allows one to investigate the spatial differences in search pattern coverage. Moon et al. use heat maps to compare the actual amount of targets in an area with sensed targets in the same area for different search methods [80]. Li et al. utilized a summary heat map to show which grid cells were visited more frequently over the course of a scenario dependent on the search pattern [49]. Lanillos et al. used 3D terrain charts representing detection probability to show how different search strategies affect the detection probability [54]. These methods work well for comparing effectiveness spatially when only varying search methods. It can be difficult, however, to attribute the differences of variations in other inputs than just the search pattern. Improving methods is needed for visualizing and understanding these differences for further exploration and analysis.

One limitation with heat map visualization techniques and refresh time metrics is they often have a large cell size, similar to the search area of the UAVs [6], which only captures a small portion of the full behavior of UAV search, showing a broad general summary of where the UAVs visited and masking specific effects of the search algorithm such as if the UAVs missed the edges of an area while searching. This is because once the cells are sufficiently small, some of the techniques used for heat maps in other research would become computationally intractable [96]. This research reintroduces a method originally presented in [93] for updating the refresh time, known in this research as the last searched time (t_{LS}), for each grid cell when the grid cells are much smaller than the UAV search/detection area in a computationally tractable way, which enables the spatial analysis techniques presented in this research.

3.4 Simulation Overview

The setup of the problem is based on the framework from the case study discussed by Day and Salmon [93], where multiple groups of UAVs work together to retrieve targets that generate according to a binomial distribution over time with an expected value (γ) for the number of targets appearing per hour. The target has an equal chance of appearing in any part of the area. The UAVs search the area and when they find targets, modeled as the target found with a circle with the UAV as the center with detection radius r_d. The UAVs then fly to the target, retrieve it, and then travel to the closest collector location and deposit it there. Upon depositing the target they return to search the area of interest. Since the UAVs have limited battery life, they return to charging stations when their state of charge is sufficiently low and recharge their battery. When the UAVs are fully charged, they take off from the charging station and resume searching. The recharging time, T_R, was set at one hour, and the flight time, T_F was set at 30 minutes to approximately match currently available technology such as the DJI Phantom 4 Pro [100]. Multiple groups of UAVs are needed to continuously cover the area because of their limited battery life, and three groups were used because of the ratio of T_R to T_F as explained in [93].

The locations for the chargers and collectors are dependent on the number of chargers and collectors in the simulation, and the configurations for each number of chargers and

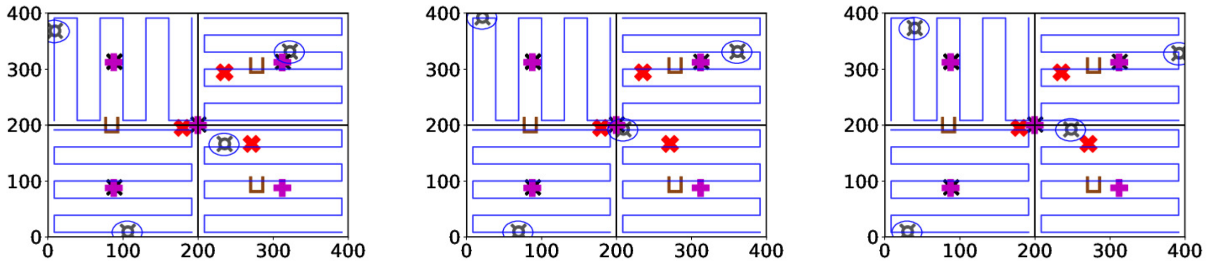
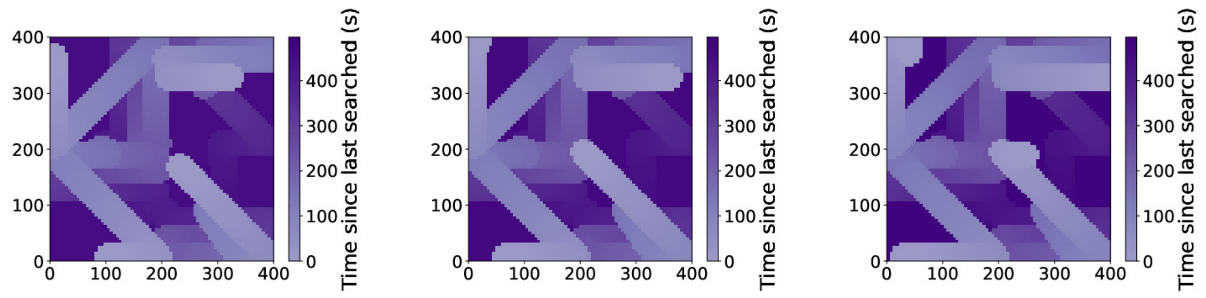

(a) $t_i = 450$ (b) $t_i = 465$ (c) $t_i = 480$

Figure 3.1: Time steps of simulation with 12 UAVs, four of them active, each patrolling in one of four partitions according to the lawnmower pattern plotted in each partition, where t_i is the time step displayed. Eight other UAVs are charging, located on charging stations. The ⌑ represents a UAV, the ⊔ represents the collectors, the ✖ represents the targets, and the ✚ represents the chargers. The UAVs have circles around them representing their target detection areas.

collectors are the same as used in [93], calculated with a differential evolution algorithm [88] with an objective function based on the target distribution model for collector placement and the probability of the UAVs losing power at a certain location for charger placement. The UAVs search according to the partitioned lawnmower pattern, as this was determined to be the best search pattern of those examined in [93]. In this pattern, the space is divided up into sections depending on the number of UAVs in the group, with each UAV patrolling one of the partitions. The UAVs each use a lawnmower pattern to search within their respective areas. Further details about the physical UAV parameters and behavior, lawnmower generation algorithm, and collector and charger placement strategies can be found in [93]. Snapshots of a simulation are shown in Fig. 3.1, where a series of time steps are shown with the UAVs following their lawnmower patterns in each partition to search for targets.

Overall effectiveness is characterized as minimizing the time that targets are in the simulation after they appear and minimizing the average number of targets in the simulation at one time. UAV search effectiveness is quantified with the average time it takes for each section of the area to be searched. To help with visual identification of search effectiveness, one can use a heat map that visualizes when areas of the map were last searched, previously discussed in [93], and referred to as the time last searched (t_{LS}) heat map. This heat map is constructed by first dividing the area of interest into square cells of equal size, with $\mathcal{G}$ being



(a) $t_i = 450$ (b) $t_i = 465$ (c) $t_i = 480$

Figure 3.2: Heat maps, each corresponding to the respective subfigure in Fig. 3.1, with each grid cell representing the amount of time since the grid cell was last searched by a UAV (t_{LS})

the set of all cells resulting from this division. For the experiments performed in this research, the maps were divided into square grid cells with a 75 × 75 grid, and therefore $\mathcal{G}$ contained 5625 grid cells. This discretization was chosen as a good balance between computational expense and detail. At each time step, cells that were not currently in the detection radius of the UAVs were increased by one (i.e. one time step), while cells in the detection radius of the UAVs were reset to zero. Cells were counted as inside the detection radius of the UAV if the inequality in Eq. 3.3 was satisfied, which is an inequality representing the Euclidian distance in grid cells from the cell wherein the UAV is located, where x_g is the horizontal number of cells away from the UAV's grid cell position, y_g is the vertical number of cells, and r_g is the grid cell radius. The equation for grid cell radius is shown in Eq. 3.1 and Eq. 3.2, where $l_{\mathcal{A}}$ is the length of the area, l_g is the length of a square grid cell, and r_d is the detection radius in unit length.

$$l_g = \frac{l_{\mathcal{A}}}{\sqrt{|\mathcal{G}|}} \tag{3.1}$$

$$r_g = Round(\frac{r_d}{l_g}) + 1 \tag{3.2}$$

$$\sqrt{(x_g^2 + y_g^2)} \leq r_g \tag{3.3}$$

Since the ratio of r_d to l_g was not usually a whole number of cells away from the UAV, it was rounded and then increment by one (i.e. radius increased by one cell) so that no cells that were in reality inside the radius would be counted as outside. This decision results in that all cells that were fully covered by the actual detection radius would be counted as searched. If this were not the case, cells that were covered could be considered missed, which gave erroneous results when trying to understand which parts of the area were not covered as often as others. More specifically, some cells would be shown as never having been searched for the whole simulation, when in reality they had been searched many times. The drawback to this approach is that some cells that were only half covered are counted as fully covered, but the discretization was small enough with a 75 $\times$ 75 grid and the partitioned lawnmower search pattern robust enough such that other small portions of half counted cells were searched. The corresponding t_{LS} heat maps to each subfigure in Fig. 3.1 are shown in Fig. 3.2. The circle representing the UAV target detection area in Fig. 3.1 is approximately discretized in Fig. 3.2, and the value of the grid cells in the UAV target detection areas are set to zero since the UAVs are currently searching that space. Since the UAVs are following a cyclical lawnmower pattern, the grid cells directly ahead of the UAV's velocity vector have the lowest values.

No matter what cell a UAV resides in, the same relative grid cells will be in range since the cell radius is the same for all grid cells (see Eq. 3.3), and so once the grid cell the UAV resides in is identified, the other grid cells in the UAV detection radius are immediately known. This is useful because no calculations are needed to know which grid cells are in a UAVs target detection area. The only calculation that must be performed is the one determining the grid cell the UAV was in, which takes a fraction of the time it would take to calculate which grid cells are in range of the UAV with a distance metric based on the actual position. This caused the metric to be computationally tractable even when the grid cells were small.

The t_{LS} heat map is important to understand because the average of these heat maps over all time steps is a good way to understand the spatial coverage for one simulation while taking into account target retrieval and delivery, as discussed in [93]. Because the average of these heat map models reveals the overall effect of UAVs pausing their search to retrieve targets, it is a good measure for understanding the spatial variance inherent in UAV search effectiveness for a single simulation run. This is opposed to metrics based purely on targets,

Table 3.2: Continuous parameters with upper and lower limits for LHS DOE

Parameter	**Lower Limit**	**Upper Limit**	**Unit**
Number of Collectors	1	10	Collectors
Number of Chargers	1	10	Chargers
Number of UAVs	6	30	UAVs
Area Length	200	800	Meters
Target Generation Rate	14.4	144.0	Targets/Hour
Target Detection Radius	10	50	Meters

which are influenced much more by randomness inherent in the simulation due to stochastic target generation.

Previously, the authors examined t_{LS} heat maps and used linear regression fitting inputs on outputs of interest to understand how multiple parameters affected outcomes, and how search patterns affected an aggregate outcome value. Three aggregate outputs were used for the previous analysis that involved the number of targets that were left out and the time it took for UAVs to search different parts of the area of interest. While these metrics were good summary indicators of effectiveness, this research steps further to understand and characterize non-linear spatial and temporal trends over time and space. A design of experiments (DOE) was created according to the latin hypercube sampling (LHS) methodology with parameter ranges shown in Tab. 3.2, used to further understand spatiotemporal patterns in multi-UAV PSR-STA. 1000 simulations were executed each with the equivalent of 3.5 days in simulation time. This period of time was chosen to guarantee that steady state conditions were reached in the vast majority of simulations.

3.5 Temporal Analysis

As stated previously, an important extension of UAV search analysis is to identify temporal trends. When exploring trends in a single simulation, simple line charts that quantify a specific metric at each time step can be effective for identifying the time steps where unusual behavior occurs in a single simulation. For use in illustrating this point, metrics from two experiments from the DOE were analyzed. The parameters of these experiments are shown in Tab. 3.3, and a snapshot of the scenarios is shown in Fig. 3.3.

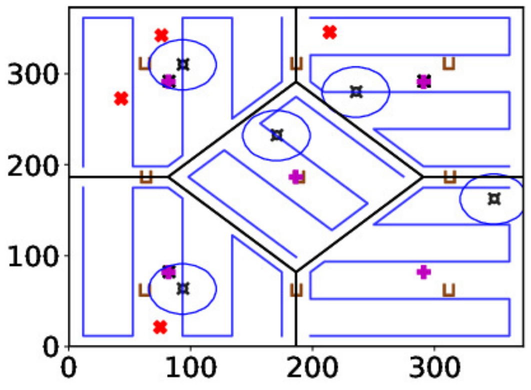
Table 3.3: Parameters for two scenarios

Parameter	**Scenario 1**	**Scenario 2**
Number of UAVs	15	25
Number of Collectors	9	1
Number of Chargers	5	10
Target Generation Rate	110.5	61.77
Target Detection Radius	27.6	12.22
Area length	373	640

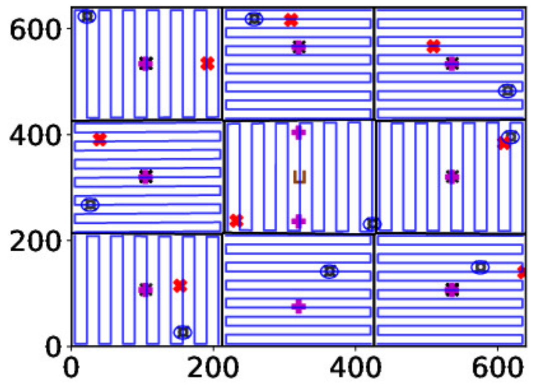
In multi-UAV PSR-STA, unusual behavior could be if a group of UAVs are not effective at searching the whole area, possibly searching one part of the area much less frequently than another, or if a target is left out for an unusually long amount of time. To identify if any target was left out for a longer than average amount of time, a line chart that records the time of the target that has been in the simulation the longest at each time step is suitable. The value of this metric at time step i is defined as $\forall t \in \mathcal{T}_i, \max(t_t)$, where $\mathcal{T}_i$ is the set of all targets at time step i, and t_t is the amount of time a target has been present in the simulation since appearing.

An example of two charts with this metric resulting from the two experiments with parameters in Tab. 3.3 is shown in Fig. 3.4. In this figure, it is apparent that scenario 2 had targets that were left out for much longer than scenario 1. This is observed because the UAVs in scenario 2 had a smaller target detection radius and a larger area than the UAVs in scenario 1, and thus required more time for the whole area to be searched. In other words, it took longer for UAVs to find a target and retrieve it in genearl once it appeared in scenario 2 than in scenario 1. For this reason, the oscillations in Fig. 3.4a are also much larger than in Fig. 3.4b.

Sometimes there were sections of the area that are not searched as often, but the targets, because of the stochastic appearance model, never appear in those sections. If only the line chart shown in Fig. 3.4 were to be examined, inefficiencies in the UAV search pattern could go undiagnosed. A line chart with the maximum t_{LS} value from $\mathcal{G}_i$, where $\mathcal{G}_i$ are the grid cells at time step i, can be used to address this concern and visualize if any parts of the area were not searched for a long time. An example of this chart is shown in Fig. 3.5b. In

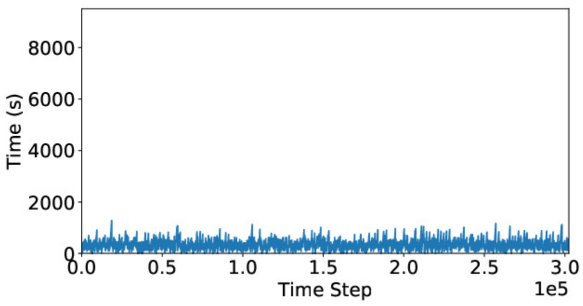


(a) Scenario 1 at $t_i = 280$

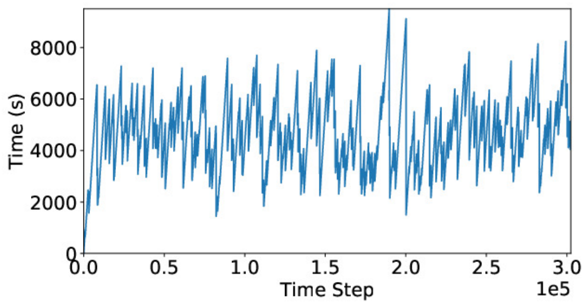


(b) Scenario 2 at $t_i = 80$

Figure 3.3: Screen capture of scenarios 1 and 2, note that in scenario 2 only the search pattern for the first group of UAVs is shown, where group one has 9 UAVs, and groups two and three have 8 UAVs



(a) Scenario 1

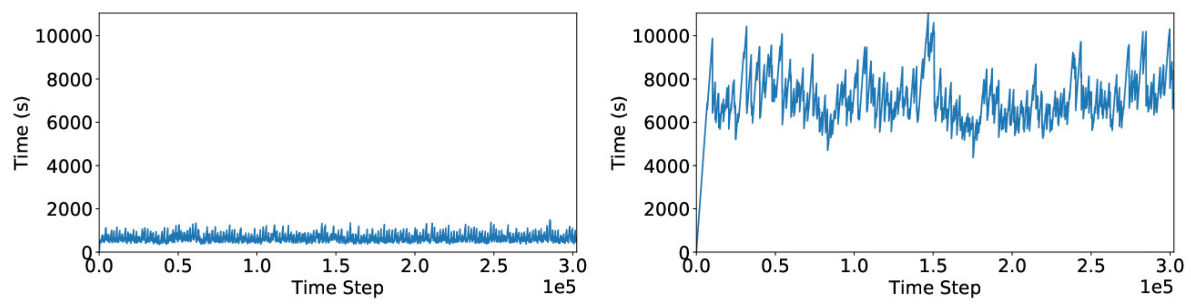


(b) Scenario 2

Figure 3.4: Maximum time that a target in $\mathcal{T}_i$ has been present

this figure, the trends are similar to Fig. 3.4, which gives confidence that these metrics are closely related for this simulation and no sections of the area are not searched as often.

From charts such as the ones displayed in Fig. 3.10, intuition can be built regarding the uniformity of the UAV search patterns. If the profile is an increasing line instead of oscillatory, this may indicate that there is one spot of the map that the UAVs never cover. This could be because there are not enough UAVs to retrieve and deposit the amount of targets that are being generated, or the search pattern does not cover part of the area. If UAVs are able to keep up with the rate of target generation and the search pattern covers every part of the area, however, the values on the chart should be oscillatory in nature. With



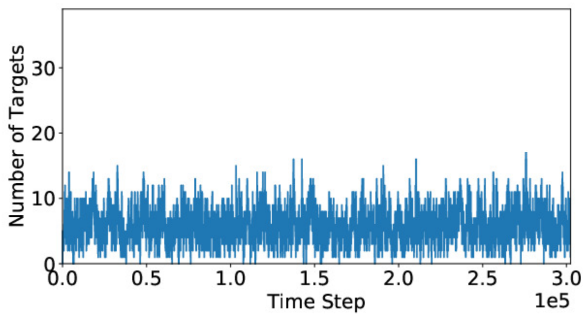
(a) Scenario 1 (b) Scenario 2

Figure 3.5: Maximum value of t_{LS} for grid cells in $\mathcal{G}_i$

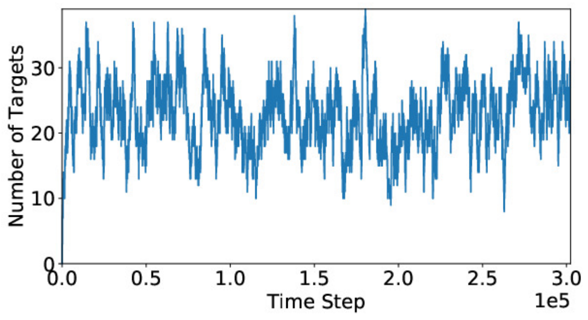
these two tools, unusual or unexpected events related to UAV search and retrieval can be spotted and examined quickly for an individual simulation and scenario.

In addition to identifying unusual events, determining which metrics are the best for understanding overall effectiveness in multi-UAV PSR-STA is equally important. In [93], the average time targets are left out, the average number of targets present, and the average value of time last searched of all grid cells over all time steps were used to understand overall effectiveness. These metrics of effectiveness are in reality summary measures of other metrics that vary over time. By examining the other metrics over time, increased insight is gained about the simulation and the original metrics of effectiveness. The average number of targets present can be examined more closely by looking at the number of targets present at each time step, shown in Fig. 3.6. Fig. 3.6b has a greater average value than Fig. 3.6a, and the deviation from the mean is also greater. The metric that can be examined to understand the average t_{LS} of all grid cells over all time steps is the average t_{LS} of the grid cells at each time step. The value of this metric at time step i is defined as $\frac{1}{|\mathcal{G}_i|}\sum_{g\in\mathcal{G}_i} t_{LS}(g)$, where $\mathcal{G}_i$ is the set of grid cells at time step i, and $t_{LS}(g)$ is the t_{LS} value of grid cell g.

This metric over time is shown in Fig. 3.7. Although the average of Fig. 3.6 increased almost fourfold, the average of Fig. 3.7 increased more than tenfold, which demonstrates that despite the area of scenario 2 is not searched as often, the lower target generation rate of scenario 2 caused the number of targets in the simulation to not increase proportionally as much as seen in Fig. 3.7.

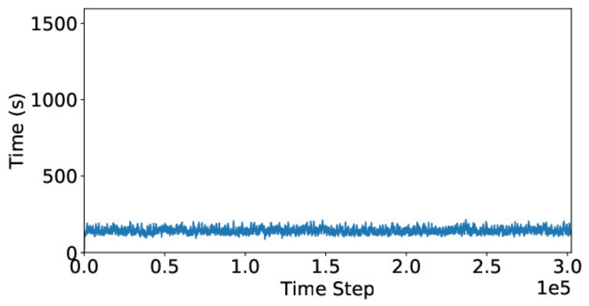


(a) Scenario 1 - Average: 5.77

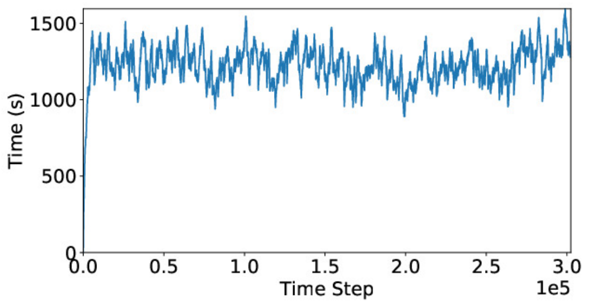


(b) Scenario 2 - Average: 22.89

Figure 3.6: Number of targets present in the simulation at each time step



(a) Scenario 1 - Average: 134.34



(b) Scenario 2 - Average: 1351.60

Figure 3.7: Average of t_{LS} for grid cells in $\mathcal{G}_i$

To verify that the steady state behavior in one simulation is representative of many scenarios, repeats of simulations with the same input parameters but different target appearance locations were performed. Three simulations were chosen from the DOE for analysis, with parameters shown in Tab. 3.3. Fig. 3.8 and 3.9 show comparisons for the number of targets in the simulation and the average of the last searched grid cell values at each time step, respectively, with output of every run superimposed on one another. In Fig. 3.8a and 3.8b, the overall oscillations of the number of targets were similar, with some small peaks from some of the simulations. Fig. 3.8c had a much higher average number of targets in the simulation than Fig. 3.8a and 3.8b, and there was more variation in the results, although the maximum values of each simulation run were in similar range bands to one another. This is important to note, as when one simulation had extreme behavior, it can be an indication that the simulation will have similar results when tested again, with a wider variation. This

Table 3.4: Parameters for experiments repeated 30 times

Parameter	**Scenario 3**	**Scenario 4**	**Scenario 5**
Number of UAVs	11	29	21
Number of Collectors	1	3	6
Number of Chargers	9	2	4
Target Detection Radius	42.78	36.32	13.8
Area Length	604	708	558
Target Generation Rate	68.68	126.55	131.46

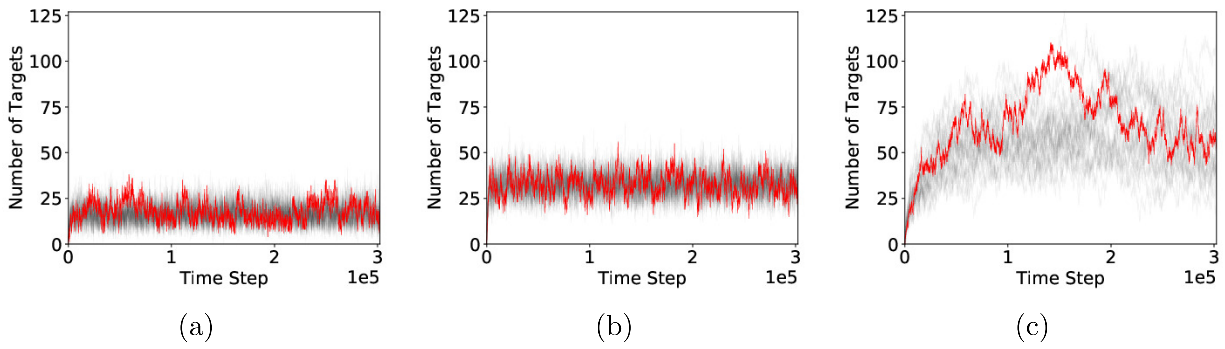
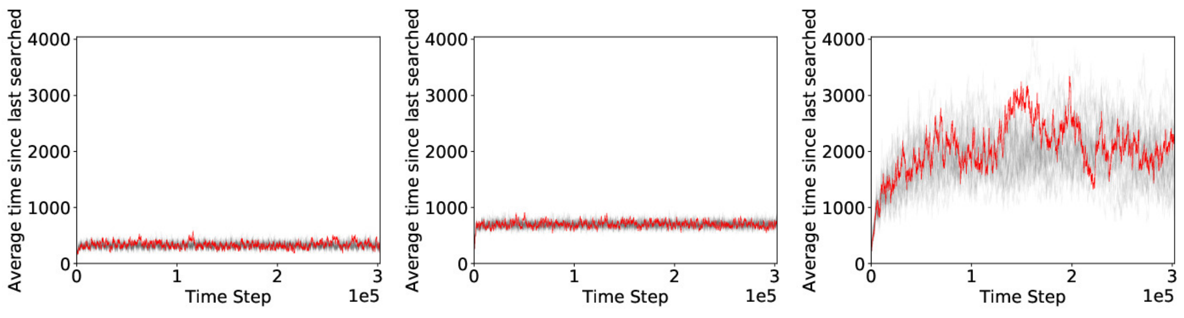


Figure 3.8: Number of targets in the simulation for three scenarios, each with 30 experiments represented by different colors, with one experiment highlighted in red to show an example scenario

is opposed to the simulation with more consistent results, which had less variation. A similar trend is shown in Fig. 3.9, but with a greater increase in variance from Fig. 3.9a and 3.9b to 3.9c. While it is time prohibitive to run all experiments 30 times, this sample provides confidence that for the situations where UAVs had a lower average number of targets present, the trends revealed in the data can be used to extrapolate to other uniform target profiles, and where simulations that perform poorly may need to be repeated.

If the UAVs' detection areas cannot completely cover the search area at every time step, which is the case for the scenarios tested in this research, there will be variation in which spaces are covered at which times. Over time, this trend can be oscillatory in nature because of the cyclical search pattern of the UAVs. Characterizing these oscillations numerically can give valuable insight into characterizing and understanding search behavior in the simulation. A strategy for doing this is by applying the discrete Fourier transform (DFT) to previously mentioned time based metrics and analyzing the results of this transform [101]. The DFT



(a) (b) (c)

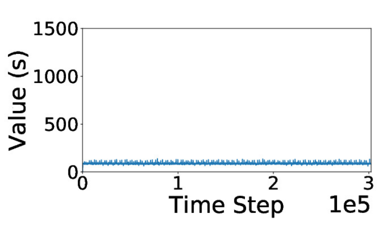
Figure 3.9: Average of t_{LS} of grid cells in $\mathcal{G}_i$ for three scenarios, each with 30 experiments represented by different colors, with one experiment highlighted in red to show an example scenario

Table 3.5: Parameters for DFT experiment analysis

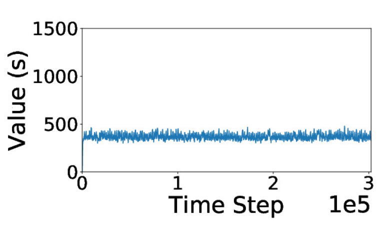
Parameter	**Scenario 6**	**Scenario 7**	**Scenario 8**
Number of UAVs	24	30	25
Number of Collectors	9	6	1
Number of Chargers	5	8	10
Target Detection Radius	14.05	14.09	12.22
Area Length	277	600	640
Target Generation Rate	45.47	57.98	61.77

decomposes a signal into a series of sine waves with different frequencies and magnitudes. If the decomposed sine waves are added together, the original signal is obtained. Because the DFT quantifies which waves that compose the signal are the largest in amplitude, the DFT can be used to identify the most significant signals that happen in the simulation. If the amplitudes of the waves are plotted along with the frequencies, the most influential ones can be easily identified and analyzed to make inferences about patterns.

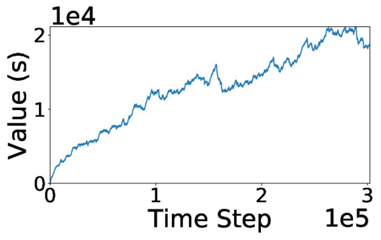
Three simulations were chosen from the DOE, with parameters shown in Tab. 3.5 to show their signals and DFT of the average of t_{LS} heat map at each time step. The DFT was calculated with the fast Fourier transform algorithm [102], implemented in the scipy package in python. These simulations were chosen as highlights of different behaviors shown across the design space. The average of t_{LS} of $\mathcal{G}_i$ was chosen as the metric to analyze for DFT, since it mitigates the effect of noise from stochastic target appearances, and makes it



(a) Average of t_{LS} of grid cells in $\mathcal{G}_i$ for 1

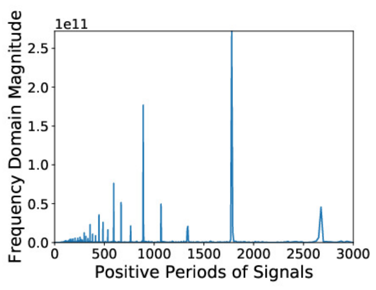


(b) Average of t_{LS} of grid cells in $\mathcal{G}_i$ for 2

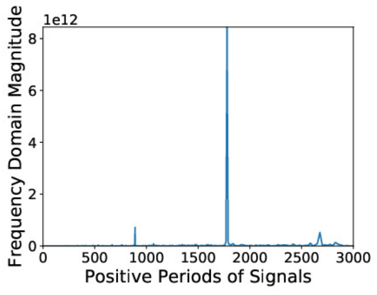


(c) Average of t_{LS} of grid cells in $\mathcal{G}_i$ for 3

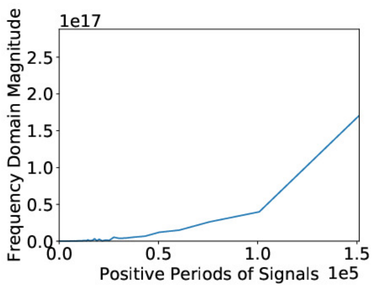
Figure 3.10: Charts describing outputs of a single simulation over time. Note that Fig. 3.10c is on a different scale since its values are a higher order of magnitude than the other figures



(a) DFT 1



(b) DFT 2



(c) DFT 3

Figure 3.11: DFT of the figures in Fig. 3.10. Note that Fig. 3.11a and 3.11b had no significant signals above 3000 and so the x-axis range was limited from 0 to 3000

easier for the DFT to identify the important temporal trends in effectiveness inherent in the simulation.

The first noticeable difference between these charts is in Fig. 3.10c, which reflects the increase in the average t_{LS} of the grid cells at each time step. This is caused because there are not enough UAVs to keep up with all the targets that are appearing, and so the UAVs use all of their time retrieving and depositing targets and never are able to explore the whole area. This is why the average value of t_{LS} of $\mathcal{G}_i$ continuously rises. In both Fig. 3.10a and 3.10b, the UAVs service the area effectively, but the total average time is slightly higher in Fig. 3.10b. This is reflected in similar signals between their DFTs, but with Fig. 3.11b having higher frequency domain magnitude than Fig. 3.11a

In Fig. 3.11a and 3.11b, there is a large signal close to 1780. This is postulated to be related to the fact that the UAV groups switch every 30 minutes, or 1800 seconds, for continuous coverage. Upon further inspection, the difference in 1800 and 1780 of 20 seconds was found to be close to the average amount of time it took for UAVs to travel from any location to a charging station at the end of their group's cycle. When the UAV groups switch, the first UAV group travels back to the chargers, and only after arrival at the chargers do the next group of UAVs fly off to search for targets. During the short time between when the first group returns and the second group starts searching there are no UAVs searching, and so the average of t_{LS} of $\mathcal{G}_i$ rises during these periods. The DFT provided an easy way to identify this increase, where it would have been more difficult to discern by only examining the time series charts in isolation. This observation brings attention to the fact that future implementations should have some overlap between the UAV group going back to charge and the next UAV group coming out to search. In this way, the second UAV group can search the area while the first UAV group travels to the chargers.

While it is convenient to study the characteristics of individual simulations, a compelling technique to understand broad trends over many simulations is by plotting the line chart output of a simulation as a data point in a figure of figures. The line charts of the 1000 experiments performed in the DOE are plotted in 25 subfigures in Fig. 3.17, with the individual smaller figures each representing one line chart on a log scale and colored based on the max value according to the legend and example presented in Fig. 3.13. The common axes are removed for clarity. Each of the 25 subfigures contains the outputs from the respective experiments classified within a subrange of target detection radii and subrange of area length, segmented into five categories as designated at the top and left of the figure of figures. Within each subfigure the x and y axis (i.e. bottom and left axes) are the number of UAVs and target generation rate, respectively. From this figure, the temporal trends in the data can be explored across four of the independent variables concurrently. The other independent variables, such as number of collectors and chargers, can likewise be used in place of the axes for additional insights. Typical outputs are plotted in figure 3.13, each overlaid one on top of the other. In this two situations can be seen to have continually increasing number of targets, and the other two are steady and oscillate in a certain range.

Use of this figure of figures, often called a trellis chart [103], is to discern general and linear trends collectively, when observing individual line charts sequentially with other

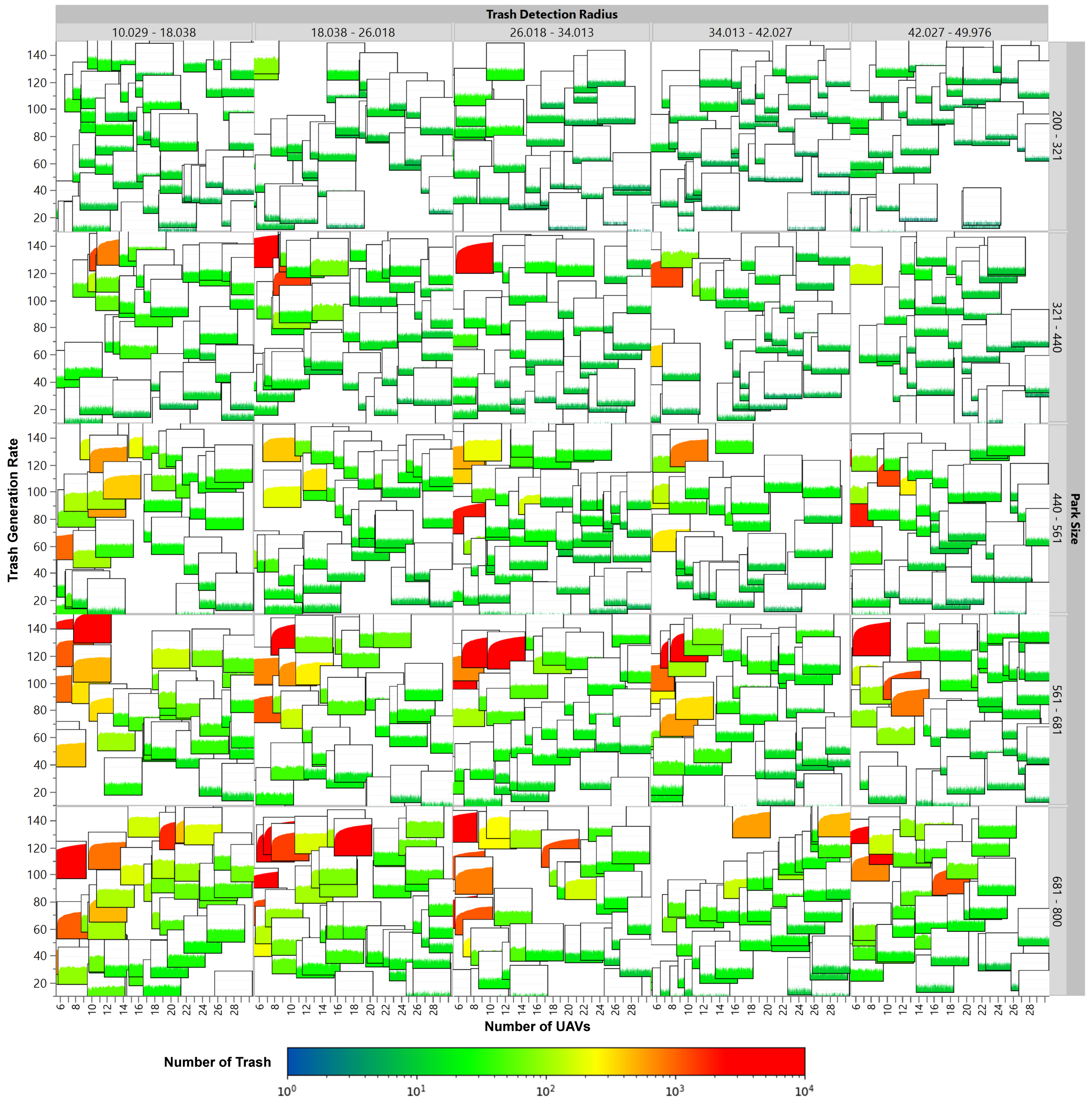


Figure 3.12: Figure of figures for the number of targets present at each time step with the simulation colored according to the maximum value in each chart

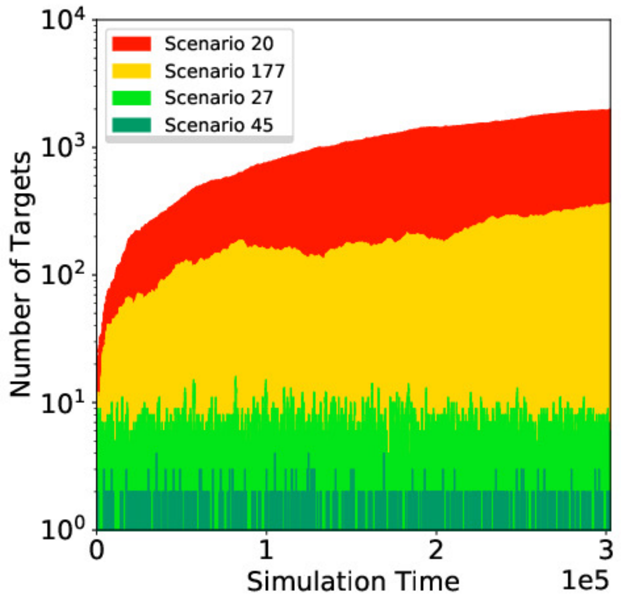


Figure 3.13: Four superimposed example figures for the subfigure icons for Fig. 3.12

means is cognitively challenging. For example, Fig. 3.17 demonstrates that as the target detection radius increases and area length decreases, the line chart values are on average lower (i.e. less targets in the simulation) than the ones with low target detection radius and high area length for different numbers of collectors and chargers. The variance of these figures also decreases with these same increases in target detection radius and decreases in area length.

3.6 Spatial Analysis

In [93], it was established that the metric for average time last searched (hereby referred to as $\overline{t_{LS}}$) is a good metric for understanding overall effectiveness in a simulation. Understanding how this metric varies spatially can bring additional understanding to how the input parameters influence overall effectiveness.

One method to gain a preliminary understanding of spatial trends is to sweep across the dimensions and explore the differences between the spatial data sets. Comparing the results to a nominal or previous output heat map, after changing an individual parameter one at a time, provides a sense for how the parameters influence spatial effectiveness, and

Table 3.6: Parameters for comparison experiments

Parameter	Baseline	Modified
Number of UAVs	12	24
Number of Collectors	3	8
Number of Chargers	3	8
Target Generation Rate	40	70
Target Detection Radius	20	50
Area length	400	700

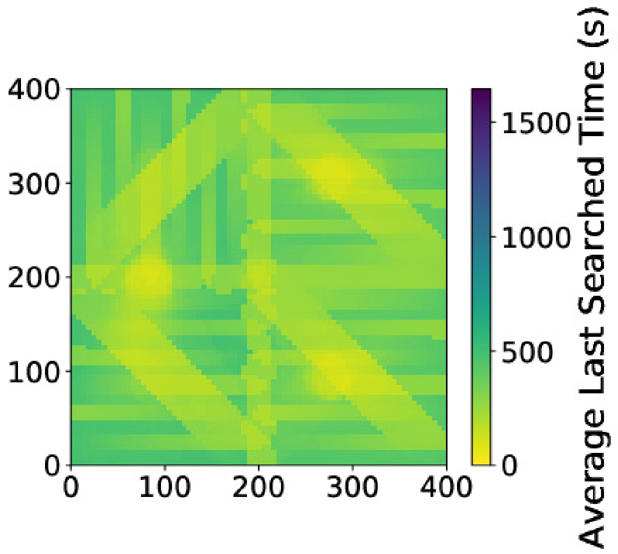
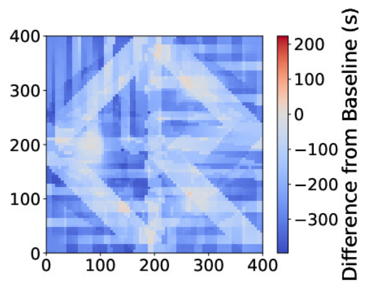


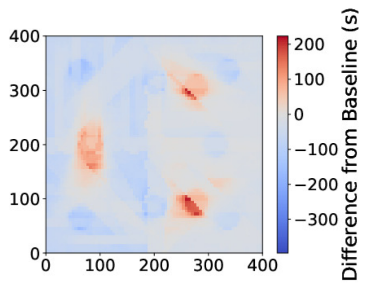
Figure 3.14: Baseline $\overline{t_{LS}}$ heat map indicates the average time that an area was last searched in seconds

differentiate which parameters affect overall effectiveness in a spatially invariant way as opposed to parameters that cause a spatially localized impact on effectiveness.

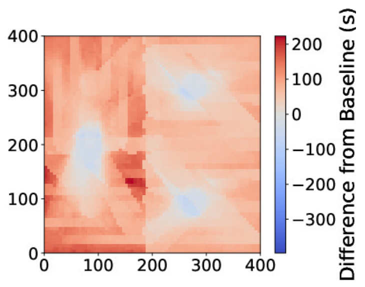
One baseline scenario, sampled from approximately the middle of the design space defined in Tab. 3.2, is compared to six other scenarios, in which a single parameter is individually varied to observe the effects on the heat map of $\overline{t_{LS}}$. The specific values of the modified parameters were chosen to be near the parameter limits of the DOE shown in Tab. 3.2, and are specified in Tab. 3.6. The baseline heat map is shown in Fig. 3.14 with the six other heat maps subtracted by the baseline heat map shown in Fig. 3.15 to more easily identify the differences and effects of parameter changes on $\overline{t_{LS}}$ with respect to this baseline scenario. Each one reveals an interesting insight about the respective parameter and



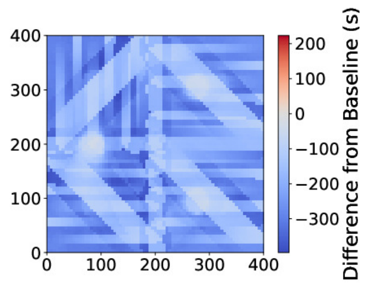
(a) UAVs = 24



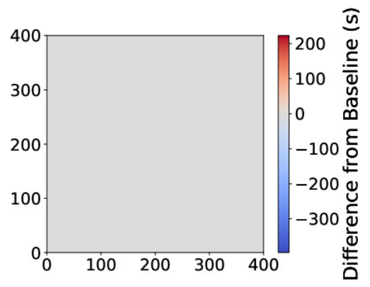
(b) Collectors = 8



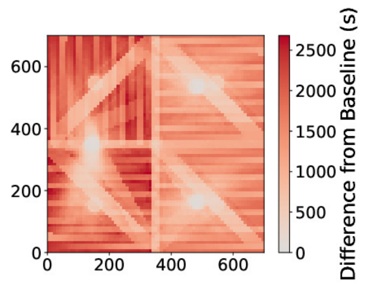
(c) $\gamma = 70$



(d) Detection radius = 50



(e) Chargers = 8



(f) Area side length = 700

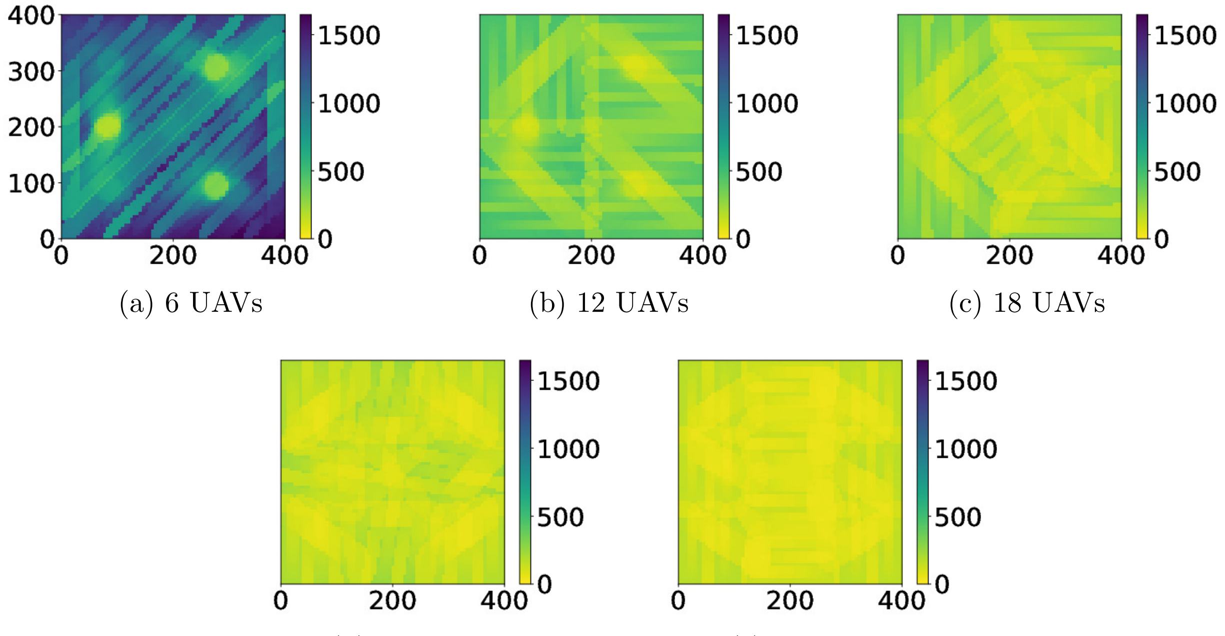
Figure 3.15: Differences from baseline experiment for $\overline{t_{LS}}$ heat map. Fig. 3.15f is on a different scale since the difference from the baseline is an order of magnitude higher than Fig. 3.15a through Fig. 3.15e

can determine the influence on the spatial patterns observed. In Fig. 3.15a, the number of UAVs was doubled from 12 to 24 resulting in a difference in the baseline heat map that was generally negative, or in other words, with a doubling of the number of UAVs, the $\overline{t_{LS}}$ was reduced, as expected. More interestingly, it also shows small spots that were slightly higher, (i.e. areas that saw an increase in the average last search time), likely due to the difference in searching patterns after more UAVs could assume smaller partitions. The key takeaway is that a non-uniform difference can be assumed from a change in the number of UAVs and that spatially the impact will not be linear across the full area of the environment. Similarly, in Fig. 3.15b, the collector positions before and after changing the number of collectors from 3 to 8 respectively are clearly shown as spots that have a positive or negative difference with respect to the baseline. The locations around the three collectors in the baseline situation are higher, since in the baseline scenario the UAVs traveled to the collectors more often. Likewise, when the number of collectors were changed to eight the UAVs instead deposited targets at the new collector locations, spreading the necessary visits across a larger number of collectors compared to the baseline's three. Because the average distance from any collector to any point in the area was decreased, the rest of the area had an overall decrease in $\overline{t_{LS}}$. The target generation increase in Fig. 3.15c caused much more traveling to the collectors, which is reflected in the decreased visit time in the areas around the collectors. The rest of the simulation, however, was not searched as often since the UAVs spent much more time retrieving and depositing targets rather than searching the space. Changing the number of chargers had negligible effects on the simulation.

When the detection radius of the UAVs increased in Fig. 3.15d, it took less time to search the whole area, decreasing $\overline{t_{LS}}$ across almost the entire area. Furthermore, since the same amount of targets appeared during the simulation with the same number of UAVs, the collector locations were visited a similar number of times resulting in a $\overline{t_{LS}}$ similar to the baseline scenario, with little or no reduction in $\overline{t_{LS}}$ at the collector locations. On the other hand, with a larger area or area length as shown in Fig. 3.15f, the UAVs take longer to travel from one part of the area to another, and the average search time is greatly increased. From comparing these difference figures, it can be seen the number of collectors and the target generation rate changes caused localized spatial effects that were most significantly related to the locations of the collectors. The spatial changes in the heat map induced by changing the target detection radius, area size, and number of UAVs were more related to the UAV search paths.

To confirm these observations for one of the input parameters (i.e. the number of UAVs), simulations were performed with the number of UAVs swept from six to 30 UAVs in increments of 6. The $\overline{t_{LS}}$ heat map for each sweep is shown in Fig. 3.16, with the same uniform scale for consistency. As identified previously, the general trend that more UAVs decreases $\overline{t_{LS}}$ overall continues. Although each number of UAVs has its own unique spatial pattern in the $\overline{t_{LS}}$ heat map correlated with the search pattern, the trend is well established that $\overline{t_{LS}}$ is consistently lowered along the sweep.

These methods are good for studying individual simulations, but another method is desired for understanding how inputs affect broad spatial trends. This can be done by plotting the heat maps of $\overline{t_{LS}}$ in a figure of figures, or trellis chart, similar to Fig. 3.12, where all 1000 simulations can be viewed concurrently at a high level. This is demonstrated in Fig.



(a) 6 UAVs (b) 12 UAVs (c) 18 UAVs

(d) 24 UAVs (e) 30 UAVs

Figure 3.16: $\overline{t_{LS}}$ heat maps of a sweep of number of UAVs from the scenarios shown in Fig. 3.14. Note that Fig. 3.16b is a repeated of Fig. 3.14 to facilitate comparisons

3.17, with the same input parameters as examined in 3.12, with each heat map presented on the same colorscale as indicated.

This figure of figures highlights several trends. First, it can be seen that increasing area length and target generation rates lead to heat map values (i.e. average last time searched) that are on average lower (i.e. shorter time to search the area on average). One non-linear relationships is that increasing the number of UAVs seems to have a greater effect on $\overline{t_{LS}}$ with smaller area lengths and larger target detection radii, than increasing the generation rate. In addition, there are distinct spatial patterns that appear, where changing parameters can decrease the last searched time in some areas more than others. This is related to the collector locations, where the locations near the collectors are searched more frequently than other areas, as discussed previously.

It is of interest to understand how different sections of the $\overline{t_{LS}}$ heat map specifically change depending on changes in every input parameter values. Although individual models for each grid cell can be used to understand how specific single cells depend on the parameter space, when there are more than 10 or 20 cells, it is difficult to comprehend any larger

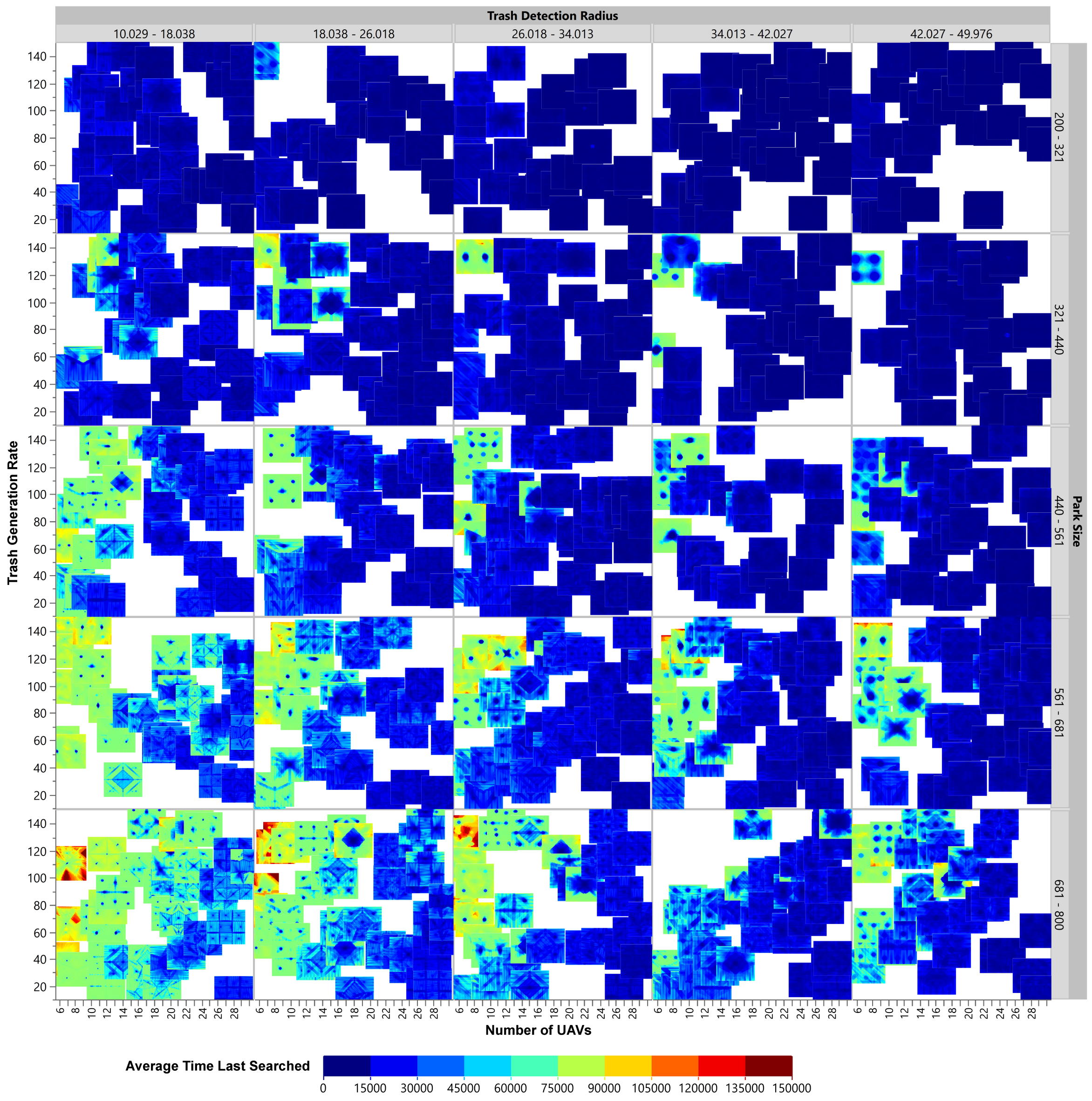
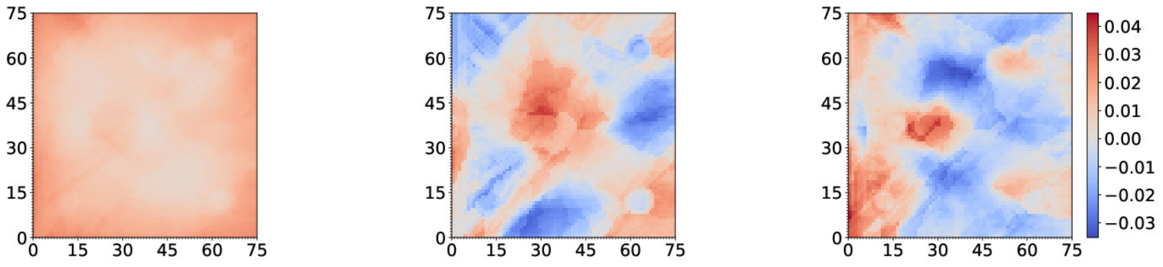


Figure 3.17: Figure of figures with the heat map of $\overline{t_{LS}}$ as each data point. A different colorscale is implemented to compare a wider range of values across the entire design space

patterns in the space. Thus, it is advantageous to identify the combination of grid cells that vary the most and are correlated together, dependent on the input parameters. Parameter reduction techniques, which identify combinations of parameters that are most relevant to the scenarios examined, is one way to bypass the limitations of models for each grid cell.

Principal component analysis (PCA) with random forest surrogate model profiling is presented in this research to describe spatial variations of $\overline{t_{LS}}$ among all the simulations



(a) PC1 - 76.2% (b) PC2 - 4.41% (c) PC3 - 3.43%

Figure 3.18: First three PCA eigenvectors of average time last searched heat maps expressed as heat maps with percent variation explained of each component in the captions

tested, and understand how input parameters affect these variations. PCA identifies sets of linear combinations of features that have the most variance in a dataset [104]. When examining the $\overline{t_{LS}}$ heat map, the features are defined as each individual $\overline{t_{LS}}$ heat map grid cell. Using PCA on a set of $\overline{t_{LS}}$ heat maps reveals which grid cells linearly vary together the most, which identifies important variational trends among all simulations. Each linear combination of features is known as a principal component (PC), and the value of how much each PC is present in an individual simulation can be quantified by a PC score [105], which with the $\overline{t_{LS}}$ heat map is calculated by multiplying the values of an individual heat map by the weightings of a PC. When the PC score is used as an output parameter in a surrogate model with explanatory input variables, then information about how the pattern defined by a PC is affected by changing the input parameters can be understood.

PCA was performed on the values of heat maps for $\overline{t_{LS}}$ for the experiments executed in the LHS DOE described in section 3.4, with each individual $\overline{t_{LS}}$ heat map included as the data points, and each grid cell $g \in \mathcal{G}$ as the features, where $\mathcal{G}$ is the set of all grid cells in the average last search heat map. Every simulation, regardless of the area size, was cast into a 75×75 grid to evaluate $\overline{t_{LS}}$, such that each simulation had the same comparable features to satisfy the requirements of PCA [106]. Visualizations of the first three principal component are shown in Fig. 3.18 with the associated percentage of variance in the design explained for each component and presented in the scree plot for the first eight PCs in Fig. 3.19.

The first principal component (PC1) accounted for 76.2 percent of the variation, and the next nine components accounted for another 19% of the variation. Because PC1

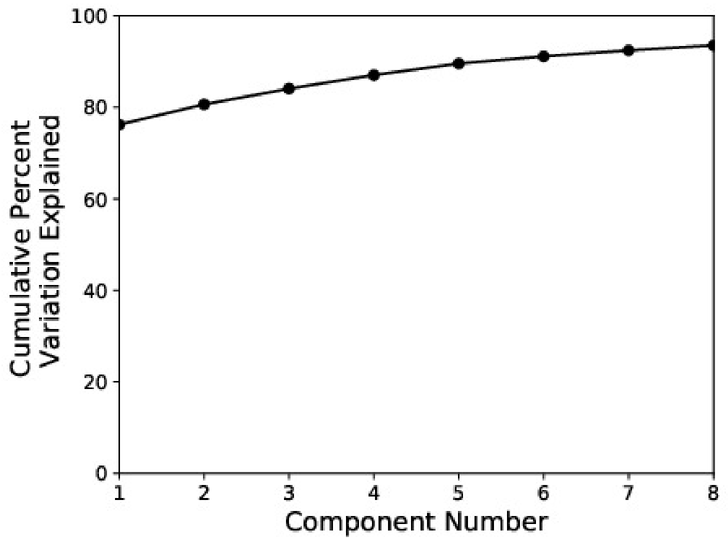


Figure 3.19: Scree plot of the cumulative percent variation of PCA for the first 8 PCs

explained so much variation, it can be concluded that PCA has significant explanatory power in relation to the dataset. If the accounted variation for a component was much less, this assumption may have been broken and other dimensionality reduction techniques should have been explored.

The general pattern for PC1, seen in Fig. 3.18a is that all parts of the map are all positively correlated. There is an important nuance to notice in this component, however, which is that the outside edges of the area have higher values than the general middle area. This means that while the value of the average last searched value raises with an increase in PC1, it is correlated with a greater increase in the edges of the area than the parts in the middle. The PC1 score for each heat map from the simulation sweep of the number of UAVs in Fig. 3.16 was calculated to understand this trend. Tab. 3.7 show the scores for each simulation. It can be observed in this table as the number of UAVs increased, the PC score decreased. However, the decrease was smaller with increasing UAVs.

To quantify the impact of the input variables on PC1, the PC score of each simulation, calculated by summing each value of the heat map multiplied by the weightings given to the linear combination of the features in PC1, can be used as an output variable and fit to a surrogate model of the input variables. This shows the influence of the input variables on PC1, where PC1 is equivalent to the magnitude of the pattern in 3.18a. The surrogate model

Table 3.7: PC scores of the average last searched heat maps from the UAV sweep displayed in Fig. 3.15

Number of UAVs	**PC score**
6	69413.63
12	22890.03
18	14669.72
24	9370.50
30	7237.79

chosen was a random forest, due to its ability to model non-linear models while also having a measure for feature importance [107]. 100 decision trees were used in the random forest. To ensure the random forest had a good fit, 9.8% of the dataset, or 98 experiments out of the 1000, were held back for validation. The distribution of the PC1 scores was approximately log normal, and so the regression was fit to log(PC1). A statistical software package, JMP, was used to generate the random forest, with the output of 100 decision trees averaged to make predictions. The R^2 of the training set was 0.994, and the R^2 of the validation set was 0.958, confirming the model possessed sufficient accuracy to use for exploratory analysis. In Tab. 3.8, measures of importance are shown for each variable. The interested reader is referred to [108] for a more detailed overview of feature importance for random forests, as it is beyond the scope of this research. The mean decrease in the sum of squares error (SSE) of an observation when the feature is used in a tree split is one important measure, with higher values equating to more explanatory power attributed to the variable. The percent column (expressed as a decimal) is the ratio of the SSE of a feature over the total SSE of all features, such that the percent columns adds to 1.0. The most important features according to these metrics were area length and number of UAVs, with other features having significant but smaller explanations of the variance.

Since the fitting function is non-linear, it cannot be said how these variables affected PC1 in a general positive or negative direction from feature importance alone. To understand trends and patterns in the data, based on input variables, a profiler tool can be used to explore how the variables affect PC1, with a snapshot of the profiler in action shown in Fig. 3.20. The purpose of a profiler is to show the trendlines of how each variable affects the output

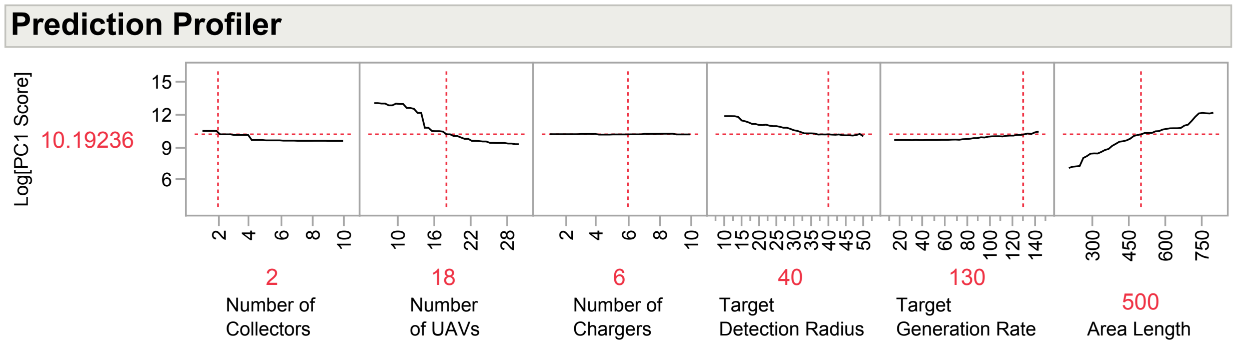


Figure 3.20: Profiler visualization

(in this case log(PC1)) assuming the other input parameters are held constant. While it is difficult to show all situations within a multi-dimensional data set, the profiler allows for exploration in design spaces that give valuable insight into non-linear behavior of variables at different parameter combinations. In Fig. 3.20, one non-linear behavior commonly observed within the scenario space was that at higher target generation and area length, there was a larger slope from 6 to 14 UAVs than the rest of domain. This is in agreement with the PC score analysis related to Tab. 3.7. The number of chargers did not have a significant effect, having a slope close to zero no matter what parameters were varied. Another observation was that after increasing the number of collectors above four, there was a smaller decrease in log(PC1) than the decrease from one to four collectors. If a cost metric were to be defined for adding UAVs, collectors, and chargers, an optimization could be performed to find the best balance between efficiency and cost for a given area with a set target generation rate and a set area length. With these tools, important spatial patterns and their influencing metrics can be examined and explored. Because of the non-linear design space, it is difficult to know every nuanced way that input parameters affect the spatial patterns, but important trends were discovered in design space regions of interest through applications of these methods.

3.7 Discussion, Limitations, and Future Work

Many broad trends were discovered throughout the analysis process. First is that for many measures of effectiveness that varied spatially over time, as the average of the measures over time increased, so did the frequency of their oscillations. This was hypothesized to have to do with the oscillatory nature of the UAV search task. One important oscillation

Table 3.8: Parameter importance for random forest model fit

Term	**SSE**	**Percent**
Area length	1141.65	0.4482
Number of UAVs	719.21	0.2823
Target Detection Radius	333.58	0.1310
Target Generation Rate	190.24	0.0747
Number of Collectors	131.29	0.0515
Number of Chargers	31.27	0.0123
Total	2547.24	1.0

discovered from the DFT analysis was that many of the simulations had a signal from DFT with a frequency close to 1780, which showed that there was a gap in searching between when one group of UAVs come back to charge and the next one was deployed. The effect of increasing UAVs, the area length, the target generation rate, and target detection radius all had differing effects on the number of targets in the simulation at each time step, with the number of UAVs being the most influential on decreasing the number of targets in the simulation at each time step.

Many spatial trends were also analyzed in this research. Through comparing to a baseline experiment, it was discovered that increasing the number of collectors and target generation rate influenced spatial patterns in effectiveness related to the collector locations, and increasing the number of UAVs, target detection radius, and area length influenced spatial patterns related to the UAV patrolling pattern. Increasing the number of chargers had a negligible influence on effectiveness. One reason this could be is because the amount of time it took for a UAV to fly anywhere in the area of interest to a charger was much less than the total flight time, which means that even if the chargers were not optimally placed, it would not affect the overall $\overline{t_{LS}}$ much. Another reason for this is because of the assumption inherent in the simulation that the chargers had enough capacity to support any amount of UAVs. If this assumption was changed, the number of chargers might have a significant effect, since if a charger was full, the UAV would need to fly farther to reach a different charger. This additional flight time could influence $\overline{t_{LS}}$ significantly, especially if the area of interest was large or non-convex.

The largest PC for the $\overline{t_{LS}}$ heat map explaining 76.2% of the variation showed that for the experiments examined, the values increased together, but it increased on the edges more than the center areas for an increase in PC1. The profiling revealed that raising PC1 was associated with an increase in target detection radius and area length, and associated with a decrease in the number of collectors and the number of UAVs, with chargers not affecting PC1, confirming previous observations. The prediction profiler revealed that the number of collectors did not make a significant difference in decreasing PC1 after more than four collectors were present in the simulation.

The analysis tools presented lead to valuable knowledge about the nature of multi-UAV PSR-STA. In the future, extensions to this research should be performed for increased understanding. In particular, studying non-square areas of interest, more complex target generation models, and uncertain target detection models will lead to further insight into multi-UAV PSR-STA. Complex search algorithms that involve real-time optimization based on these extensions should also be employed to increase UAV search effectiveness. Furthermore, tests with actual UAVs should be performed to validate these results.

3.8 Conclusion

This research presented spatial and temporal analysis on an implementation of multi-UAV PSR-STA. Measures were highlighted which provided insight into performance variability over time, visualized in line charts, for a given simulation, and DFT was used to further understand the temporal patterns inherent in the data. The trellis chart or figure of figures method was presented for visualizing spatial and temporal data across the full design space with many simulations. PCA was used to find the relevant spatial patterns inherent over the simulations, and the random forest method with a profiler were used to explore the non-linear influence of input parameters on the spatial patterns. These highlight some methodologies and metrics for analyzing PSR-STA beyond simple aggregate values, and served to increase understanding about which factors influence the effectiveness of UAV search in multi-UAV PSR-STA.

CHAPTER 4. CONCLUSIONS

The intent of this thesis was to create a framework that builds a foundation for understanding how to simulate and analyze multi-UAV PSR-STA, prescribing important design decisions and methods for simulation, and identifying metrics and analysis tools for understanding overall system effectiveness. Through fulfilling this intent, this thesis provides understanding about design decisions and analysis methods that allow for the simulation and analysis of real-world multi-UAV PSR-STA scenarios. The four outcomes of this thesis, proposed in the introduction of this thesis, outline the process taken to understand and analyze multi-UAV PSR-STA, fulfilling the intent of this thesis. These outcomes were to:

1. Propose a framework that facilitates simulation design through identifying design decisions that should be made to successfully simulate multi-UAV PSR-STA

2. Implement a simulation model and necessary algorithms for successful study of multi-UAV PSR-STA, including a method for placement of chargers and collectors dependent on probabilistic information

3. Identify important metrics to characterize system effectiveness of multi-UAV PSR-STA and identify trends related to these metrics

4. Examine many different simulations of PSR-STA to verify the usefulness of the framework, metrics, and methods developed as a result of previous outcomes

In Chapter 2 a framework for simulating and analyzing the multi-UAV PSR-STA was presented and discussed, addressing the first outcome. This framework presented important design decisions for simulating multi-UAV PSR-STA, summarized in Fig. 2.1. An analysis framework was also presented which identified two factors, UAV search effectiveness and the influence of the amount of resources in a simulation, as important to analyze for understanding system behavior. These frameworks pinpointed which areas require focus for effective simulation and analysis of multi-UAV PSR-STA, fulfilling outcome one.

In the implementation of this framework in Chapter 2, unique algorithms and metrics of effectiveness were developed. An algorithm for charger and collector placement based on probabilistic information was developed. A general state diagram for UAV behavior was introduced, along with relevant equations that specified UAV behavior that satisfied the operational requirements for servicing PSR-STA. This fulfilled outcome two.

Three metrics were introduced in Chapter 2 that quantified effectiveness of a simulation through assessing UAV search performance and measuring target statistics. Another metric was introduced, visualized by a heat map, which allowed for insight into the spatial variation in multi-UAV search coverage. A case study was executed, with comparison testing of four search patterns within the constraints of the framework. Statistical methods examining the UAV search effectiveness metric showed the partitioned lawnmower search pattern performed the best compared to other search patterns, and the influence of various parameters on overall effectiveness metrics suggested that increasing the number of UAVs is, initially, the best choice to increase system effectiveness over increasing charger or collector locations for typical park sizes. The global lawnmower pattern was found to have certain deficiencies that should be addressed for optimal coverage. Through these analytical insights and introduction of unique metrics, outcome three was addressed, while the simulation and examination of various scenarios to discover these insights addressed outcome four.

In Chapter 3, additional metrics that further quantified temporal and spatial trends were demonstrated, which provided insight into performance variability over time and space respectively. Temporal analysis measures were highlighted which provided insight into performance variability over time, visualized in line charts, for a given simulation, and the discrete Fourier transform was used to further understand the temporal patterns present in the data. Principal component analysis was used to find the relevant spatial patterns in UAV search effectiveness inherent over the simulations, and the random forest surrogate model with a profiler was used to explore the non-linear influence of input parameters on the spatial patterns. The trellis figure of figures method was presented for visualizing spatial and temporal data across many simulations. Chapter 3 highlighted some methodologies and metrics for analyzing multi-UAV PSR-STA, and served to increase understanding about which factors influence the effectiveness of UAV search in multi-UAV PSR-STA, further addressing outcomes three and four.

Through accomplishing outcomes one through four, a useful foundation of knowledge was developed for simulating and analyzing multi-UAV PSR-STA. By understanding the important design decisions, one can understand what assumption are required to successfully simulate multi-UAV PSR-STA. By understanding relevant metrics and analyzing those metrics with a variety of analysis techniques, one can gain an understanding of how to thoroughly analyze multi-UAV PSR-STA.

4.1 Limitations and Future Work

One of the limitations of this study is that the scenarios presented were confined to an agent-based simulation. This thesis did not perform live performance tests with real UAVs, which could have served as a powerful validation test for the usefulness of this framework. While this methodology revealed many preliminary insights about UAV search patterns, more research should be completed with real-world tests to gain insight on multi-UAV PSR-STA. The trends identified and information gained from this study, however, are valuable for future realistic testing, and can provide preliminary inputs for decisions regarding which UAV search strategies are most promising to test.

The assumptions in the scenarios tested were valid for the situations tested, but changes in the assumptions could have changed the results of the analyses. The landing, taking off, retrieving targets, and depositing targets were all modeled as constant time, but they could also be modeled as non-constant time tasks, with the time changing depending on the task performed. These assumptions could be adjusted to characterize the sensitivity of model results to the time to perform these tasks. The UAV motion model was also simple, chosen to reduce computational cost for the ability to simulate a larger number of scenarios, which was necessary for some of the analysis techniques introduced. More complex motion models could be implemented for additional understanding of the effect of complex motion models on UAV search effectiveness.

An extension to the scenarios tested is to analyze scenarios with areas of interest that are non-square shaped. This extension would require a UAV search pattern that could cover non-convex areas, but the collector and charger placement algorithm would function the same, with the placement being limited to inside the area. Another extension to this problem is considering non-uniform target appearance models. In the research performed as part of this thesis, the target appearance model was a binomial distribution, with a uniform probability model, which allowed the use of lawnmower coverage patterns for UAVs since targets had an equal chance of appearing anywhere in the area of interest. With a non-uniform probability model, such as two independent normal distributions, or a model that matches real-world behavior such as the littering tendencies of people at a particular park, the problem domain could become more complex. In the case of park littering, agent-based models of littering tendencies would need to be developed, perhaps from sociological studies. Different classifications of people could be identified, such as bicyclists and pedestrians, and their littering characteristics defined in the agent-based simulation. The UAVs could then search and collect the litter while these agents are present in the park. Other important issues to address when human agents are involved are to generate a strategy for human avoidance and decide how the UAVs will discern between litter and a person's belongings.

With these additions the probability map of where targets appear might not be known initially, and so the UAVs could learn the probability distribution of the target appearance that is resultant from any of these additions, and adjust their search patterns accordingly. They could also adjust their search patterns at different times of day depending on if the target appearance probability changes throughout the day. Some types of litter, such as plastic bags, could be moved by the wind or other environmental factors, and this movement could be included in the probabilistic target appearance model, influencing the multi-UAV search patterns.

The search pattern would have to be adjusted in these situations to ensure that the UAVs search areas that have a higher probability of target appearance more often than others. There are some space transformation techniques that could be used to address these non-uniform cases, involving stretching the lawnmower pattern to cover more important areas more often than others [109]. However if the probability distribution is highly discontinuous or varies throughout time, further research is needed, especially in the context of multi-UAV search, since a space transformation technique may not be sufficient to cover the areas proportionally according to importance. If partitioning the area for collision-free multi-UAV search, the partitions should take into account the probability map by including equal probabilities in each respective partition or by splitting high probability areas among many partitions. As mentioned previously, these methods should also account for the changing target generation patterns that will be present at different times of day. Further verification tests of simulations, analyzed with the analysis tools presented in this thesis, can be used to judge the efficacy of these methods.

Along with non-uniform target appearance models, adding probabilistic detection models to the problem is another area for exploration. Most image recognition algorithms have a rate of false positives [110], and so this would have to be taken into account in the simulation when searching. This would change the primary metric used to identify spatial patterns in this thesis from the last searched time, t_{LS}, to one relating to probability, such as the probability of a target existing in the area. This would need to be updated at each time step using a Bayesian update for the grid cells in a UAVs detection area taking into account rates of false positives [96], and a different update would be performed for grid cells outside the detection area.

Another important multi-UAV search pattern that could be implemented in these situations is an algorithm that makes real-time decisions about where to search instead of relying on pre-computed paths like the lawnmower coverage pattern. This algorithm could use probabilistic information about the likelihood of targets existing in certain areas to make decisions. One such option is the receding horizon control [111], where each UAV looks a number of time steps into the future and decides on the best path to take depending on an objective function. The advantage of these kinds of methods is that UAVs can take into account many complicated factors related to path planning that are captured in an objective function. These complications arise when the practical problems related to multi-UAV PSR-STA become more intricate and the simulation of multi-UAV PSR-STA increases in fidelity. However, real-time optimization methods are computationally expensive, and so would decrease the number of simulations able to be performed, which could inhibit the ability to analyze multi-UAV PSR-STA for broad trends over a wide range of scenarios.

Allowing the UAVs to remember the locations of previously detected targets and to communicate this information with other UAVs should also be considered. Including these features introduces many interesting challenges. Consider the case where one UAV sees four targets in its local area, and another UAV is searching in another area with no targets present. Should the second UAV join the first UAV and help it collect the targets, or should it continue to search in case more targets appear in its area? This is also known as the exploration-exploitation tradeoff [20]. These dilemmas arise especially when the target appearance model is unknown or is highly discontinuous. One strategy could be to have some UAVs assigned to searching for targets, and other types of UAVs or ground robots assigned to target retrieval, and through this strategy UAV search would not be interrupted. Further research should be done to address these concerns in the context of multi-UAV PSR-STA.

A major difficulty arising in optimized search and consensus algorithms is that in many of the algorithms, there are many tunable parameters that can influence UAV effectiveness, but are difficult to choose since the outcome of changing the parameters cannot be easily predicted. One use of the analysis methods presented in this thesis could be to understand how changes in parameters affect the performance of the UAVs through time and space for a wide range of scenarios. This could be useful when trying to deploy UAVs for a task in various locations, as one combination of algorithm parameters could make the UAVs more effective in one scenario, such as in a smaller area, whereas if the same parameter combination was used in another scenario, adverse effects could occur such as the UAVs missing the corners of an area. The analysis techniques demonstrated in this thesis could be used to tune parameters and find different sets of parameters suitable for various situations, as opposed to using a single set of parameters for all situations.

4.2 Final Remarks

This thesis introduced a framework that outlined design decisions, analysis metrics, and methods for simulating and analyzing multi-UAV PSR-STA. Through the framework, the initial hurdles of understanding the assumptions and design decisions that need to be considered to simulate multi-UAV PSR-STA were overcome. Overall, the framework is a useful tool as an initial reference in understanding the unique challenges that come with simulating and analyzing multi-UAV PSR-STA. As UAVs gain additional functionality for interacting with their environment and become more ubiquitous, multi-UAV PSR-STA will gain importance as an area to be studied and understood. The spatial and temporal analysis methods presented in this research will become increasingly useful, since with the complexity of deploying UAVs in the modern world, detailed spatiotemporal information will be required to understand and implement multi-UAV PSR-STA into various real-world scenarios.

There are many areas where multi-UAV PSR-STA will be applicable in the future as technology advances, including search and rescue after a disaster, where UAVs must search for and retrieve people to relocate them to a safe location after a disaster has occurred, and litter cleanup, where UAVs search for litter to retrieve and deposit it in a trash bin. As referenced in the introduction of this thesis, creating solutions to the problems and challenges related to these areas is important and would improve the lives of many people. Though theses problems and challenges do not have simple solutions, an effective approach to solve them can stem from applying the framework and analysis methods for multi-UAV PSR-STA introduced in this thesis to the area of interest. Through fulfilling the objective of this thesis and creating a framework and analysis methods that can be applied to generate solutions to real-world problems, another step is taken to better the world with the aid of UAVs.

REFERENCES

[1] Goodrich, M. A., Morse, B. S., Gerhardt, D., Cooper, J. L., Quigley, M., Adams, J. A., and Humphrey, C., 2008. "Supporting wilderness search and rescue using a camera-equipped mini UAV." *Journal of Field Robotics,* **25**(1-2), pp. 89–110. 1

[2] Mader, D., Blaskow, R., Westfeld, P., and Weller, C., 2016. "Potential of UAV-based laser scanner and multispectral camera data in building inspection." *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences,* **XLI-B1**, pp. 1135–1142. 1

[3] Manyam, S. G., Rasmussen, S., Casbeer, D. W., Kalyanam, K., and Manickam, S., 2017. "Multi-UAV routing for persistent intelligence surveillance & reconnaissance missions." In *2017 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE, pp. 573–580. 1

[4] Nahar, P., Wu, K.-h., Mei, S., Ghoghari, H., Srinivasan, P., Lee, Y.-l., Gao, J., and Guan, X., 2017. "Autonomous UAV forced graffiti detection and removal system based on machine learning." In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, IEEE, pp. 1–8. 1

[5] Faiçal, B. S., Freitas, H., Gomes, P. H., Mano, L. Y., Pessin, G., [de Carvalho], A. C., Krishnamachari, B., and Ueyama, J., 2017. "An adaptive approach for UAV-based pesticide spraying in dynamic environments." *Computers and Electronics in Agriculture,* **138**, pp. 210 – 223. 1

[6] Nigam, N., 2014. "The multiple unmanned air vehicle persistent surveillance problem: A review." *Machines,* **2**(1), Jan, p. 13–72. 2, 7, 10, 37, 39

[7] Wei, C., Hindriks, K. V., and Jonker, C. M., 2016. "Dynamic task allocation for multi-robot search and retrieval tasks." *Applied Intelligence,* **45**(2), pp. 383–401. 2, 7, 9

[8] Rangoni, R., and Jager, W., 2017. "Social dynamics of littering and adaptive cleaning strategies explored using agent-based modelling." *Journal of Artificial Societies and Social Simulation,* **20**(2). 2, 35

[9] Chiang, C.-H., 2015. "Vision-based coverage navigation for robot trash collection task." In *2015 International Conference on Advanced Robotics and Intelligent Systems (ARIS)*, IEEE, pp. 1–6. 2, 35

[10] Bai, J., Lian, S., Liu, Z., Wang, K., and Liu, D., 2018. "Deep learning based robot for automatically picking up garbage on the grass." *IEEE Transactions on Consumer Electronics,* **64**(3), pp. 382–389. 2

[11] Stickel, B. H., Jahn, A., and Kier, B., 2012. The cost to West Coast communities of dealing with trash, reducing marine debris. Prepared by Kier Associates for US Environmental Protection Agency, Region 9, pursuant to Order for Services EPG12900098. San Francisco, CA. 2, 35

[12] Scherer, J., and Rinner, B., 2016. "Persistent multi-UAV surveillance with energy and communication constraints." In *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, IEEE, pp. 1225–1230. 2, 8, 10

[13] Shakhatreh, H., Khreishah, A., Chakareski, J., Salameh, H. B., and Khalil, I., 2016. "On the continuous coverage problem for a swarm of UAVs." In *2016 IEEE 37th Sarnoff Symposium*, IEEE, pp. 130–135. 2, 8, 10

[14] Li, B., Moridian, B., Kamal, A., Patankar, S., and Mahmoudian, N., 2019. "Multi-robot mission planning with static energy replenishment." *Journal of Intelligent & Robotic Systems,* **95**(2), pp. 745–759. 2, 8, 10, 36

[15] Park, H., and Morrison, J. R., 2019. "System design and resource analysis for persistent robotic presence with multiple refueling stations." In *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE, pp. 622–629. 2, 8, 10

[16] Ranque, P., Freeman, D., Kernstine, K., Lim, D., Garcia, E., and Mavris, D. *Stochastic Agent-Based Analysis of UAV Mission Effectiveness*. 3, 10, 35, 37, 38

[17] Ruiwen, Z., Bifeng, S., Yang, P., and Qijia, Y., 2019. "Improved method for subsystems performance trade-off in system-of-systems oriented design of UAV swarms." *Journal of Systems Engineering and Electronics,* **30**(4), pp. 720–737. 3, 35, 38

[18] Muratore, M., Silvestrini, R. T., and Chung, T. H., 2014. "Simulation analysis of UAV and ground teams for surveillance and interdiction." *The Journal of Defense Modeling and Simulation,* **11**(2), pp. 125–135. 3, 35, 38

[19] Ravichandran, R., Ghose, D., and Das, K., 2019. "UAV based survivor search during floods." In *2019 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE, pp. 1407–1415. 3, 35

[20] Jin, Y., Liao, Y., Minai, A. A., and Polycarpou, M. M., 2005. "Balancing search and target response in cooperative unmanned aerial vehicle (UAV) teams." *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics),* **36**(3), pp. 571–587. 3, 35, 70

[21] Alamdari, S., Fata, E., and Smith, S. L., 2014. "Persistent monitoring in discrete environments: Minimizing the maximum weighted latency between observations." *The International Journal of Robotics Research,* **33**(1), pp. 138–154. 3, 10

[22] Ding, Y., Luo, W., and Sycara, K., 2019. "Decentralized multiple mobile depots route planning for replenishing persistent surveillance robots." In *2019 International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, IEEE, pp. 23–29. 3, 38

[23] Shakhatreh, H., Sawalmeh, A. H., Al-Fuqaha, A., Dou, Z., Almaita, E., Khalil, I., Othman, N. S., Khreishah, A., and Guizani, M., 2019. "Unmanned aerial vehicles (UAVs): A survey on civil applications and key research challenges." *IEEE Access,* **7**, pp. 48572–48634. 7, 34

[24] Shakeri, R., Al-Garadi, M. A., Badawy, A., Mohamed, A., Khattab, T., Al-Ali, A. K., Harras, K. A., and Guizani, M., 2019. "Design challenges of multi-UAV systems in cyber-physical applications: A comprehensive survey and future directions." *IEEE Communications Surveys & Tutorials,* **21**(4), pp. 3340–3385. 7

[25] Coffey, T., and Montgomery, J. A., 2002. "The emergence of mini UAVs for military applications." *Defense Horizons*(22). 7

[26] Winfield, A. F., 2009. "Towards an engineering science of robot foraging." In *Distributed Autonomous Robotic Systems 8*. Springer, pp. 185–192. 9

[27] Bayındır, L., 2016. "A review of swarm robotics tasks." *Neurocomputing,* **172**, pp. 292–321. 9

[28] Zedadra, O., Jouandeau, N., Seridi, H., and Fortino, G., 2017. "Multi-agent foraging: state-of-the-art and research challenges." *Complex Adaptive Systems Modeling,* **5**(1), p. 3. 9

[29] Darmanin, R. N., and Bugeja, M. K., 2017. "A review on multi-robot systems categorised by application domain." In *2017 25th Mediterranean Conference on Control and Automation (MED)*, IEEE, pp. 701–706. 9

[30] Pitonakova, L., Crowder, R., and Bullock, S., 2016. "Information flow principles for plasticity in foraging robot swarms." *Swarm Intelligence,* **10**(1), pp. 33–63. 9

[31] Brambilla, M., Ferrante, E., Birattari, M., and Dorigo, M., 2013. "Swarm robotics: a review from the swarm engineering perspective." *Swarm Intelligence,* **7**(1), pp. 1–41. 9

[32] Pitonakova, L., Crowder, R., and Bullock, S., 2018. "Information exchange design patterns for robot swarm foraging and their application in robot control algorithms." *Frontiers in Robotics and AI,* **5**, p. 47. 9

[33] Bähnemann, R., Schindler, D., Kamel, M., Siegwart, R., and Nieto, J., 2017. "A decentralized multi-agent unmanned aerial system to search, pick up, and relocate objects." In *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, IEEE, pp. 123–128. 9

[34] Spurný, V., Báča, T., Saska, M., Pěnička, R., Krajník, T., Thomas, J., Thakur, D., Loianno, G., and Kumar, V., 2019. "Cooperative autonomous search, grasping, and

delivering in a treasure hunt scenario by a team of unmanned aerial vehicles." *Journal of Field Robotics,* **36**(1), pp. 125–148. 9

[35] Loianno, G., Spurny, V., Thomas, J., Baca, T., Thakur, D., Hert, D., Penicka, R., Krajnik, T., Zhou, A., Cho, A., et al., 2018. "Localization, grasping, and transportation of magnetic objects by a team of MAVs in challenging desert-like environments." *IEEE Robotics and Automation Letters,* **3**(3), pp. 1576–1583. 9

[36] Stump, E., and Michael, N., 2011. "Multi-robot persistent surveillance planning as a vehicle routing problem." In *2011 IEEE International Conference on Automation Science and Engineering*, IEEE, pp. 569–575. 9

[37] Semsch, E., Jakob, M., Pavlicek, D., and Pechoucek, M., 2009. "Autonomous UAV surveillance in complex urban environments." In *2009 IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology*, Vol. 2, IEEE, pp. 82–85. 9

[38] Matai, R., Singh, S. P., and Mittal, M. L., 2010. "Traveling salesman problem: an overview of applications, formulations, and solution approaches." *Traveling salesman problem, theory and applications,* **1**. 9

[39] Bektas, T., 2006. "The multiple traveling salesman problem: an overview of formulations and solution procedures." *Omega,* **34**(3), pp. 209–219. 9

[40] Cabreira, T., Brisolara, L., and R Ferreira, P., 2019. "Survey on coverage path planning with unmanned aerial vehicles." *Drones,* **3**(1), p. 4. 9

[41] Kadioglu, E., Urtis, C., and Papanikolopoulos, N., 2019. "UAV coverage using hexagonal tessellation." In *2019 27th Mediterranean Conference on Control and Automation (MED)*, IEEE, pp. 37–42. 9

[42] Palacios-Gasós, J. M., Talebpour, Z., Montijano, E., Sagüés, C., and Martinoli, A., 2017. "Optimal path planning and coverage control for multi-robot persistent coverage in environments with obstacles." In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, IEEE, pp. 1321–1327. 9

[43] Maza, I., and Ollero, A., 2007. "Multiple UAV cooperative searching operation using polygon area decomposition and efficient coverage algorithms." In *Distributed Autonomous Robotic Systems 6*. Springer, pp. 221–230. 9

[44] Bentz, W., and Panagou, D., 2018. "Energy-aware persistent coverage and intruder interception in 3D dynamic environments." In *2018 Annual American Control Conference (ACC)*, IEEE, pp. 4426–4433. 10

[45] Trotta, A., Di Felice, M., Montori, F., Chowdhury, K. R., and Bononi, L., 2018. "Joint coverage, connectivity, and charging strategies for distributed UAV networks." *IEEE Transactions on Robotics,* **34**(4), pp. 883–900. 10

[46] Gainer Jr, J., Dawkins, J., DeVries, L., and Kutzer, M., 2019. "Persistent multi-agent search and tracking with flight endurance constraints." *Robotics,* **8**(1), p. 2. 10, 37

[47] Erdelj, M., Saif, O., Natalizio, E., and Fantoni, I., 2019. “UAVs that fly forever: Uninterrupted structural inspection through automatic UAV replacement.” *Ad Hoc Networks,* **94**, p. 101612. 10

[48] Hartuv, E., Agmon, N., and Kraus, S., 2019. “Scheduling spare drones for persistent task performance with several replacement stations.” In *2019 International Symposium on Multi-Robot and Multi-Agent Systems (MRS)*, IEEE, pp. 95–97. 10, 12

[49] Li, B., Patankar, S., Moridian, B., and Mahmoudian, N., 2018. “Planning large-scale search and rescue using team of UAVs and charging stations.” In *2018 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, IEEE, pp. 1–8. 10, 11, 36, 37, 38

[50] Ribeiro, R. G., Júnior, J. R., Cota, L. P., Euzébio, T. A., and Guimarães, F. G., 2019. “Unmanned aerial vehicle location routing problem with charging stations for belt conveyor inspection system in the mining industry.” *IEEE Transactions on Intelligent Transportation Systems.* 10

[51] Palacios-Gasós, J. M., Montijano, E., Sagüés, C., and Llorente, S., 2016. “Distributed coverage estimation and control for multirobot persistent tasks.” *IEEE transactions on Robotics,* **32**(6), pp. 1444–1460. 10, 38

[52] Singh, S., Lu, S., Kokar, M. M., Kogut, P. A., and Martin, L., 2017. “Detection and classification of emergent behaviors using multi-agent simulation framework (WIP).” In *Proceedings of the Symposium on Modeling and Simulation of Complexity in Intelligent, Adaptive and Autonomous Systems*, MSCIAAS ’17, Society for Computer Simulation International. 10

[53] Smith, S. L., and Rus, D., 2010. “Multi-robot monitoring in dynamic environments with guaranteed currency of observations.” In *49th IEEE conference on decision and control (CDC)*, IEEE, pp. 514–521. 10

[54] Lanillos, P., Gan, S. K., Besada-Portas, E., Pajares, G., and Sukkarieh, S., 2014. “Multi-UAV target search using decentralized gradient-based negotiation with expected observation.” *Information Sciences,* **282**, pp. 92 – 110. 10, 37, 38

[55] Yu, J., Karaman, S., and Rus, D., 2015. “Persistent monitoring of events with stochastic arrivals at multiple stations.” *IEEE Transactions on Robotics,* **31**(3), pp. 521–535. 10, 36, 37

[56] Szafir, D. A., 2018. “The good, the bad, and the biased: five ways visualizations can mislead (and how to fix them).” *Interactions,* **25**(4), pp. 26–33. 11

[57] Li, Z., Zhu, C., and Gold, C., 2004. *Digital terrain modeling: principles and methodology.* CRC press. 12

[58] Enright, J. J., Frazzoli, E., Pavone, M., and Savla, K., 2015. “UAV routing and coordination in stochastic, dynamic environments.” *Handbook of unmanned aerial vehicles*, pp. 2079–2109. 12, 14

[59] Enright, J., Frazzoli, E., Savla, K., and Bullo, F., 2005. "On multiple UAV routing with stochastic targets: Performance bounds and algorithms." In *Aiaa guidance, navigation, and control conference and exhibit*, p. 5830. 12

[60] Seyedi, S., Yazicioğlu, Y., and Aksaray, D., 2019. "Persistent surveillance with energy-constrained UAVs and mobile charging stations." *IFAC-PapersOnLine,* **52**(20), pp. 193–198. 12

[61] Jung, S., and Ariyur, K. B., 2017. "Automated wireless recharging for small UAVs." *International Journal of Aeronautical and Space Sciences,* **18**(3), pp. 588–600. 12

[62] Ure, N. K., Chowdhary, G., Toksoz, T., How, J. P., Vavrina, M. A., and Vian, J., 2014. "An automated battery management system to enable persistent missions with multiple aerial vehicles." *IEEE/ASME transactions on mechatronics,* **20**(1), pp. 275–286. 12, 13

[63] Boukoberine, M. N., Zhou, Z., and Benbouzid, M., 2019. "A critical review on unmanned aerial vehicles power supply and energy management: Solutions, strategies, and prospects." *Applied Energy,* **255**, p. 113823. 12

[64] Hassanalian, M., and Abdelkefi, A., 2017. "Classifications, applications, and design challenges of drones: A review." *Progress in Aerospace Sciences,* **91**, pp. 99–131. 12

[65] Dubins, L. E., 1957. "On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents." *American Journal of Mathematics,* **79**(3), pp. 497–516. 13, 16

[66] Wu, L., Ke, Y., and Chen, B. M., 2018. "Systematic modeling of rotor-driving dynamics for small unmanned aerial vehicles." *Unmanned Systems,* **6**(02), pp. 81–93. 13

[67] Beard, R. W., and McLain, T. W., 2012. *Small unmanned aircraft: Theory and practice.* Princeton university press. 13

[68] Cocchioni, F., Frontoni, E., Ippoliti, G., Longhi, S., Mancini, A., and Zingaretti, P., 2016. "Visual based landing for an unmanned quadrotor." *Journal of Intelligent & Robotic Systems,* **84**(1-4), pp. 511–528. 13

[69] Villa, D. K., Brandão, A. S., and Sarcinelli-Filho, M., 2019. "A survey on load transportation using multirotor UAVs." *Journal of Intelligent & Robotic Systems*, pp. 1–30. 13

[70] Parra-Vega, V., Sanchez, A., Izaguirre, C., Garcia, O., and Ruiz-Sanchez, F., 2013. "Toward aerial grasping and manipulation with multiple UAVs." *Journal of Intelligent & Robotic Systems,* **70**(1-4), pp. 575–593. 13

[71] Symington, A., Waharte, S., Julier, S., and Trigoni, N., 2010. "Probabilistic target detection by camera-equipped UAVs." In *2010 IEEE International Conference on Robotics and Automation*, IEEE, pp. 4076–4081. 13

[72] Radmanesh, M., Kumar, M., Guentert, P. H., and Sarim, M., 2018. "Overview of path-planning and obstacle avoidance algorithms for UAVs: A comparative study." *Unmanned systems,* **6**(02), pp. 95–118. 13

[73] Lin, Z., Castano, L., Mortimer, E., and Xu, H., 2020. "Fast 3D collision avoidance algorithm for fixed wing UAS." *Journal of Intelligent & Robotic Systems,* **97**(3), pp. 577–604. 13

[74] Zhang, L., Wang, J., Lin, Z., Lin, L., Chen, Y., and He, B., 2019. "Distributed cooperative obstacle avoidance for mobile robots using independent virtual center points." *Journal of Intelligent & Robotic Systems*, pp. 1–15. 13

[75] Torabbeigi, M., Lim, G. J., and Kim, S. J., 2020. "Drone delivery scheduling optimization considering payload-induced battery consumption rates." *Journal of Intelligent & Robotic Systems,* **97**(3), pp. 471–487. 13

[76] Dietrich, T., Krug, S., and Zimmermann, A., 2017. "An empirical study on generic multicopter energy consumption profiles." In *2017 Annual IEEE International Systems Conference (SysCon)*, IEEE, pp. 1–6. 13

[77] Prasetia, A. S., Wai, R.-J., Wen, Y.-L., and Wang, Y.-K., 2019. "Mission-based energy consumption prediction of multirotor UAV." *IEEE Access,* **7**, pp. 33055–33063. 13

[78] Vasquez-Gomez, J. I., Marciano-Melchor, M., Valentin, L., and Herrera-Lozada, J. C., 2020. "Coverage path planning for 2D convex regions." *Journal of Intelligent & Robotic Systems,* **97**(1), pp. 81–94. 14

[79] Mitchell, D., Chakraborty, N., Sycara, K., and Michael, N., 2015. "Multi-robot persistent coverage with stochastic task costs." In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, IEEE, pp. 3401–3406. 14

[80] Moon, B. G., and Peterson, C. K., 2018. "Learned search parameters for cooperating vehicles using gaussian process regressions." In *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, IEEE, pp. 493–502. 14, 38

[81] Khamis, A., Hussein, A., and Elmogy, A., 2015. "Multi-robot task allocation: A review of the state-of-the-art." In *Cooperative Robots and Sensor Networks 2015*. Springer, pp. 31–51. 14

[82] Bethke, B., How, J., and Vian, J., 2009. "Multi-UAV persistent surveillance with communication constraints and health mangement." In *AIAA Guidance, Navigation, and Control Conference*, p. 5654. 14

[83] Hansen, S., McLain, T., and Goodrich, M., 2007. "Probabilistic searching using a small unmanned aerial vehicle." In *AIAA Infotech@ Aerospace 2007 Conference and Exhibit*, p. 2740. 16

[84] Bidstrup, C. C., Moore, J. J., Peterson, C. K., and Beard, R. W., 2019. "Tracking multiple vehicles constrained to a road network from a UAV with sparse visual

measurements." In *2019 American Control Conference (ACC)*, IEEE, pp. 3817–3822. 16

[85] 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), pp. 445–462. 20

[86] Dobrin, A., 2005. "A review of properties and variations of voronoi diagrams." *Whitman College*, pp. 1949–3053. 20

[87] Galceran, E., and Carreras, M., 2013. "A survey on coverage path planning for robotics." *Robotics and Autonomous systems,* **61**(12), pp. 1258–1276. 20

[88] Storn, R., and Price, K., 1997. "Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces." *Journal of global optimization,* **11**(4), pp. 341–359. 24, 40

[89] Kraft, D., 1994. "Algorithm 733: TOMP–fortran modules for optimal control calculations." *ACM Transactions on Mathematical Software (TOMS),* **20**(3), pp. 262–281. 25

[90] Forrester, A., Sobester, A., and Keane, A., 2008. *Engineering design via surrogate modelling: a practical guide*. John Wiley & Sons. 29

[91] Abdi, H., and Williams, L. J., 2010. "Tukey's honestly significant difference (HSD) test." *Encyclopedia of Research Design. Thousand Oaks, CA: Sage*, pp. 1–5. 31

[92] Tsouros, D. C., Bibi, S., and Sarigiannidis, P. G., 2019. "A review on UAV-based applications for precision agriculture." *Information,* **10**(11), p. 349. 34

[93] Day, R., and Salmon, J. A framework for multi-UAV persistent search and retrieval with stochastic target appearance submitted. 34, 36, 39, 40, 42, 46, 53

[94] Acevedo, J. J., Arrue, B. C., Maza, I., and Ollero, A., 2015. "Distributed cooperation of multiple UAVs for area monitoring missions." In *Motion and Operation Planning of Robotic Systems*. Springer, pp. 471–494. 37

[95] George, J., Sujit, P., and Sousa, J. B., 2011. "Search strategies for multiple UAV search and destroy missions." *Journal of Intelligent & Robotic Systems,* **61**(1-4), pp. 355–367. 37

[96] Waharte, S., Symington, A., and Trigoni, N., 2010. "Probabilistic search with agile UAVs." In *2010 IEEE International Conference on Robotics and Automation*, IEEE, pp. 2840–2845. 37, 39, 69

[97] Ramasamy, M., and Ghose, D., 2017. "A heuristic learning algorithm for preferential area surveillance by unmanned aerial vehicles." *Journal of Intelligent & Robotic Systems,* **88**(2-4), pp. 655–681. 37

[98] Zhang, R., Song, B., Pei, Y., Tang, W., and Wang, M., 2017. "Agent-based analysis of multi-UAV area monitoring mission effectiveness." In *AIAA Modeling and Simulation Technologies Conference*, p. 3151. 37

[99] Saeed, A., Abdelkader, A., Khan, M., Neishaboori, A., Harras, K. A., and Mohamed, A., 2019. "On realistic target coverage by autonomous drones." *ACM Transactions on Sensor Networks (TOSN),* **15**(3), pp. 1–33. 38

[100] Phantom, D. pro/pro+ user manual. 2018 `https://dl.djicdn.com/downloads/phantom_4_pro/Phantom+4+Pro+Pro+Plus+User+Manual+v1.0.pdf` Accessed: 2020-06-29. 39

[101] Jaffe, D. A., 1987. "Spectrum analysis tutorial, part 1: the discrete Fourier transform." *Computer Music Journal,* **11**(2), pp. 9–24. 48

[102] Cooley, J. W., and Tukey, J. W., 1965. "An algorithm for the machine calculation of complex Fourier series." *Mathematics of computation,* **19**(90), pp. 297–301. 49

[103] Cleveland, W. S., 1993. *Visualizing data.* Hobart Press. 51

[104] Song, F., Guo, Z., and Mei, D., 2010. "Feature selection using principal component analysis." In *2010 international conference on system science, engineering design and manufacturing informatization*, Vol. 1, IEEE, pp. 27–30. 59

[105] Jolliffe, I. T., 1990. "Principal component analysis: a beginner's guide—i. introduction and application." *Weather,* **45**(10), pp. 375–382. 59

[106] Wold, S., Esbensen, K., and Geladi, P., 1987. "Principal component analysis." *Chemometrics and intelligent laboratory systems,* **2**(1-3), pp. 37–52. 59

[107] Liaw, A., Wiener, M., et al., 2002. "Classification and regression by randomForest." *R news,* **2**(3), pp. 18–22. 61

[108] Grömping, U., 2009. "Variable importance assessment in regression: linear regression versus random forest." *The American Statistician,* **63**(4), pp. 308–319. 61

[109] Nolan, P., Paley, D. A., and Kroeger, K., 2017. "Multi-UAS path planning for non-uniform data collection in precision agriculture." In *2017 IEEE Aerospace Conference*, IEEE, pp. 1–12. 68

[110] Gaszczak, A., Breckon, T. P., and Han, J., 2011. "Real-time people and vehicle detection from UAV imagery." In *Intelligent Robots and Computer Vision XXVIII: Algorithms and Techniques*, Vol. 7878, International Society for Optics and Photonics, p. 78780B. 69

[111] Yao, P., Wang, H., and Ji, H., 2017. "Gaussian mixture model and receding horizon control for multiple UAV search in complex environment." *Nonlinear Dynamics,* **88**(2), pp. 903–919. 69

APPENDIX A. CODE

A.1 Simulation Code

run_park_sim.py

```python
from time import time
import random
import sys

from parkcleanup.parkcleanup.simulation.park_cleanup_simulation import ParkCleanupSimulation
from parkcleanup.parkcleanup.builders.sim_model_builder import SimModelBuilder
from parkcleanup.parkcleanup.builders.drone_builder import DroneBuilder
from parkcleanup.parkcleanup.visualization.matplotlib_plotter import MatplotlibPlotter
from parkcleanup.parkcleanup.dataloggers.sim_data_logger import SimDataLogger
from collector_placement_algorithms.placement_data_utils import load_avgmin_config

def main():
    # This script sets up a simulation and runs it
    print("Starting simulation calculations")
    start = time()
    random.seed(55555)

    bounds = 300
    num_collectors = 8
    num_chargers = 5
    collector_coords = load_avgmin_config(num_collectors, bounds).tolist()
    charging_station_coords = load_avgmin_config(num_chargers, bounds).tolist()

    trash_per_hour = 150
    trash_spawn_rate = trash_per_hour/3600
    sim_model_builder = (
        SimModelBuilder()
        .set_park_bounds(bounds)
        .init_collectors(collector_coords)
        .init_rechargers(charging_station_coords)
        .set_random_trash_generation_on(trash_spawning_rate=trash_spawn_rate)
    )
    drone_builder = (
        DroneBuilder(bounds)
        .set_starting_position_random()
        .set_speed(3)
        .set_fly_time(1800)
        .set_recharge_time(3600)
        .set_trash_detection_radius(20)
        .set_object_found_distance(3)
        .set_constant_trash_dropoff_delay(5)
        .set_constant_trash_pickup_delay(5)
        .set_charging_params(
            set_out_for_seen_trash_while_charging=1.0,
            emergency_recharge_level=0.05,
            return_to_charge_from_patrolling=0.05
        )
        .set_number_of_drones_to_init(15)
        .set_starting_position_on_coordinates(charging_station_coords)
        .set_start_delay()
        .set_search_method_partitioned_lawnmower()
    )
    drones = drone_builder.commit()

    sim_model_builder.init_drones(drones)
    sim_model = sim_model_builder.commit()
    sim = (
        ParkCleanupSimulation(sim_model)
    )

    sim.run_sim(total_time_steps=5000, data_logger=SimDataLogger(10,75, False))
    end = time()

    # Set plotting settings
    plotter = (
        MatplotlibPlotter()
        .show_trash_detection_radius_circle()
        .set_drone_color_change_for_battery_level()
        .show_drone_search_patterns()
        .show_outputs()
    )

    print("Time from initialization to model calculations: " + str(end-start))
    plotter.interactive_plot_data(sim)

if __name__ == "__main__":
    main()
```

park_cleanup_simulation.py

```
from time import time
import random
import sys
from random import random as rand
from copy import copy

from scipy.spatial import distance_matrix
import numpy as np

from parkcleanup.parkcleanup.model.agents.person import Person
from parkcleanup.parkcleanup.model.agents.drone import Drone
from parkcleanup.parkcleanup.model.objectives.collector import Collector
from parkcleanup.parkcleanup.model.objectives.trash import Trash
from parkcleanup.parkcleanup.tools.helper import sign
from parkcleanup.parkcleanup.model.agents.drone import DroneStateType

class ParkCleanupSimulation:
    def __init__(self, sim_model):
        self.sim_model = sim_model
        self.num_time_steps = None
        self.random_seed = None
        self._sim_has_finished = False
        self.trash_id_counter = 0

    def run_sim(self, total_time_steps, data_logger=None, seed_for_run=None):
        if self._sim_has_finished:
            raise Exception("Simulation has already been run")
        if seed_for_run is None:
            seed_for_run = random.randrange(sys.maxsize)
        random.seed(seed_for_run)
        self.random_seed = seed_for_run
        num_time_steps = total_time_steps
        self.num_time_steps = num_time_steps

        if data_logger is not None:
            self.data_logger = data_logger
            data_logger.update_initial_information(self)

        self._initialize_drone_states()
        for index in range(0, num_time_steps):
            self.sim_model.curr_time_step = index
            self._step()
            if data_logger is not None:
                data_logger.update(index, self.sim_model)
        self._sim_has_finished = True
        data_logger.update_final_information(self)

    def has_run(self):
        return self._sim_has_finished

    def _initialize_drone_states(self):
        # This allows the drones to set themselves up based on the sim_model
        for drone in self.sim_model.all_drones:
            drone._set_state(drone._state_type, self.sim_model)

    def _step(self):
        if self.sim_model.persons_on:
            self._update_persons()
        self._update_drones()
        self._update_trash()

    def _update_trash(self):
        for trash in self.sim_model.all_trash:
            trash.time_left_out += 1
        if self.sim_model.random_trash_generation_on:
            self._randomly_generate_trash()

    def _calculate_distance_to_drop_off(self, x, y):
        distances_from_collectors = distance_matrix(self.sim_model.collector_coords, [[x, y]])
        closest_collector_distance = min(distances_from_collectors).item(0)
        closest_collector = self.sim_model.collector_coords[np.argmin(distances_from_collectors)]
        distances_from_chargers = distance_matrix(self.sim_model.charger_coords, [closest_collector])
        closest_charger_distance = min(distances_from_chargers)
        # Add 3 for safety buffer
        return closest_charger_distance + closest_collector_distance + 30 + 3

    def _randomly_generate_trash(self):
        if rand() < self.sim_model.trash_spawning_rate:
            random_x = rand()*self.sim_model.park.bounds
            random_y = rand()*self.sim_model.park.bounds
            distance_to_drop_off = self._calculate_distance_to_drop_off(random_x, random_y)
            new_trash = Trash([random_x,random_y], distance_to_drop_off, self.sim_model.curr_time_step,
 self.trash_id_counter)
            self.trash_id_counter += 1
            self.sim_model.all_trash.append(new_trash)

    def _update_persons(self):
        if rand() < self.sim_model.person_spawning_rate:
            speed = self.sim_model.person_params[0]
            trash_percent_threshold = self.sim_model.person_params[1]
            found_distance = self.sim_model.person_params[2]
            max_path = self.sim_model.person_params[3]
            new_person = Person(speed, trash_percent_threshold, found_distance, self.sim_model.park,
 max_path)
            self.sim_model.all_persons.append(new_person)
            self.sim_model.data_logger.total_persons += 1
        for index, person in enumerate(self.sim_model.all_persons):
            person.update()
            if person.throws_trash():
                distance_to_drop_off = self._calculate_distance_to_drop_off(*person.position)
                self.sim_model.all_trash.append(Trash(person.position, distance_to_drop_off))
            if person.finished():
                del self.sim_model.all_persons[index]
                if not self.sim_model.all_persons:
                    self.sim_model.all_persons = []

    def _update_drones(self):
        self.sim_model.update_drone_info()
        # Update the drone objectives
        for drone in self.sim_model.all_drones:
            drone.update(self.sim_model)
```

sim_model.py

```
from scipy.spatial import distance_matrix

class SimModel(object):
    '''
    The purpose of this class is to store the state of the simulation at each
    time step. It should not be populated with historical data that grow over
    time so that the simulation can be run with a near constant amount of RAM if
    desired. Data logging should be delagated to another class.
    '''
    def __init__(self):
        # Use SimModelBuilder for initialization
        self.all_drones = None
        self.all_persons = []
        self.all_trash = []

        self.random_trash_generation_on = None
        self.trash_spawning_rate = None

        self.persons_on = None
        self.person_params = None
        self.person_spawning_rate = None

        self.all_collectors = None
        self.all_drones = None
        self.all_rechargers = None

        self.park = None

        self.drone_coords = None
        self.trash_coords = None
        self.person_coords = None
        self.collector_coords = None
        self.charger_coords = None
        self.drone_to_drone = None
        self.drone_to_person = None
        self.drone_to_trash = None
        self.drone_to_collector = None

        self.times_left_out = None
        self.times_left_out_positions = None
        self.trash_ids = None
        self.start_times = None

        self.curr_time_step = None
        self.potential_fields_on = None

    def update_drone_info(self):
    # Find all distances to objects around the drones
        drone_coords = [drone.position for drone in self.all_drones]
        person_coords = []
        trash_coords = []
        drone_to_trash = []
        drone_to_person = []
        drone_to_drone = []
        if self.potential_fields_on:
            drone_to_drone = distance_matrix(drone_coords, drone_coords).tolist()
        if len(self.all_persons) != 0:
            person_coords = [person.position for person in self.all_persons]
            drone_to_person = distance_matrix(drone_coords, person_coords).tolist()
        if self.there_is_trash_in_model():
            trash_coords = [trash.position for trash in self.all_trash]
            drone_to_trash = distance_matrix(drone_coords, trash_coords).tolist()
            for index, trash in enumerate(self.all_trash):
                trash.distances_to_drones = [one_drone_to_all_trash[index] for one_drone_to_all_trash in
    drone_to_trash]
        self.update_temp_info(drone_coords, trash_coords, person_coords, drone_to_drone, drone_to_person,
     drone_to_trash)

    def update_temp_info(self, drone_coords, trash_coords, person_coords, drone_to_drone, drone_to_person
    , drone_to_trash):
        self.drone_coords = drone_coords
        self.trash_coords = trash_coords
        self.person_coords = person_coords
        self.drone_to_drone = drone_to_drone
        self.drone_to_person = drone_to_person
        self.drone_to_trash = drone_to_trash
        self.times_left_out = []
        self.times_left_out_positions = []
        self.trash_ids = []
        self.start_times = []

    def record_trash_pickup_event(self, trash):
        self.trash_ids.append(trash.id)
        self.start_times.append(trash.start_time)
        self.times_left_out.append(trash.time_left_out)
        self.times_left_out_positions.append(trash.position)

    def there_is_trash_in_model(self):
        return len(self.all_trash) != 0

    def there_are_people_in_model(self):
        return len(self.all_persons) != 0

    def drones_have_trash(self):
        has_trash = [drone.has_trash for drone in self.all_drones]
        return (True in has_trash)
```

sim_model_builder.py

```
from random import random as rand

from parkcleanup.parkcleanup.simulation.sim_model import SimModel
from parkcleanup.parkcleanup.model.objectives.collector import Collector
from parkcleanup.parkcleanup.model.objectives.charge_station import ChargeStation
from parkcleanup.parkcleanup.model.park.park import Park
from parkcleanup.parkcleanup.tools.helper import random_position_in_bounds
from collector_placement_algorithms.placement_data_utils import load_avgmin_config

class SimModelBuilder(object):
    def __init__(self):
        self._random_trash_generation_on = None
        self._trash_spawning_rate = None
        self._park_bounds = None
        self._person_params = None
        self._person_spawning_rate = None
        self._persons_on = None
        self._all_collectors = None
        self._all_rechargers = None
        self._all_drones = None

    def set_random_trash_generation_on(self, trash_spawning_rate):
        if trash_spawning_rate < 0 or trash_spawning_rate > 1.0:
            raise ValueError("Trash spawning rate not in range")
        self._random_trash_generation_on = True
        self._trash_spawning_rate = trash_spawning_rate
        return self

    def set_persons_on(self, walking_speed, litter_rate, found_objective_distance, num_paths_to_walk,
     spawning_rate):
        if litter_rate > 1.0 or litter_rate < 0:
            raise ValueError("Person litter rate must be between zero and one")
        if found_objective_distance <= 0:
            raise ValueError("Found objective distance must be positive and nonzero")
        if walking_speed <= 0:
            raise ValueError("Walking speed must be positive and non zero")
        if not isinstance(num_paths_to_walk, int) and not num_paths_to_walk.is_integer():
            raise TypeError("Paths to walk must be int")
        if num_paths_to_walk < 1:
            raise ValueError("Num paths to walk must be positive and non-zero")
        self._person_params = [walking_speed, litter_rate, found_objective_distance, num_paths_to_walk]
        self._person_spawning_rate = spawning_rate
        self._persons_on = True
        return self

    def set_park_bounds(self, bounds):
        if bounds <= 0:
            raise ValueError("Park bounds is not in range")
        self._park_bounds = bounds
        return self

    def init_drones(self, drones):
        self._all_drones = drones
        return self

    def init_collectors(self, start_positions):
        if self._park_bounds is None:
            raise Exception("Park bounds must be set before initializing collectors")
        all_collectors = []
        all_collector_coords = []
        for coords in start_positions:
            self._check_coords(coords)
            all_collector_coords.append(coords)
            all_collectors.append(Collector(coords))
        self._all_collectors = all_collectors
        self._all_collector_coords = all_collector_coords
        return self

    def init_rechargers_from_file(self, num_chargers, bounds):
        all_chargers = load_avgmin_config(num_chargers, bounds).tolist()
        self.init_rechargers(all_chargers)
        return self

    def init_collectors_from_file(self, num_collectors, bounds):
        collector_coords = load_avgmin_config(num_collectors, bounds).tolist()
        self.init_collectors(collector_coords)
        return self

    def init_rechargers_random(self, num_chargers):
        charging_station_coords = []
        for _ in range(num_chargers):
            charging_station_coords.append(random_position_in_bounds(self._park_bounds))
        self.init_rechargers(charging_station_coords)
        return self

    def init_collectors_random(self, num_collectors):
        collector_coords = []
        for _ in range(num_collectors):
            collector_coords.append(random_position_in_bounds(self._park_bounds))
        self.init_collectors(collector_coords)
        return self

    def init_rechargers(self, start_positions):
        if self._park_bounds is None:
            raise Exception("Park bounds must be set before initializing chargers")
        all_chargers = []
        all_chargers_coords = []
        for coords in start_positions:
            self._check_coords(coords)
            all_chargers.append(ChargeStation(coords))
            all_chargers_coords.append(coords)
        self._all_rechargers = all_chargers
        self._all_recharger_coords = all_chargers_coords
        return self

    def _check_coords(self, coords):
        self._check_coords_type(coords)
        self._check_that_coords_are_in_bounds(coords)

    def _check_coords_type(self, coords):
        if len(coords) != 2 or not isinstance(coords[0], (int, float)) or not isinstance(coords[1], (int,
     float)):
                raise TypeError("Coordinate location must be list of length two with float or int")

    def _check_that_coords_are_in_bounds(self, coords):
        if coords[0] < 0 or coords[0] > self._park_bounds or coords[1] < 0 or coords[1] > self.
    _park_bounds:
            raise ValueError("Coordinate location must be in park bounds")

    def commit(self):
        if self._random_trash_generation_on is None:
            self._random_trash_generation_on = False
        if self._persons_on is None:
            self._persons_on = False
        self._check_if_can_commit()
        sim_model = SimModel()
        self._set_sim_model_parameters(sim_model)
        self._set_drone_ids(sim_model)
        self._set_collector_ids(sim_model)
        self._set_charger_ids(sim_model)
        return sim_model

    def _check_if_can_commit(self):
        if self._all_collectors is None or len(self._all_collectors) == 0:
            raise Exception("No collectors in simulation")
        if self._all_rechargers is None or len(self._all_rechargers) == 0:
            raise Exception("No rechargers in simulation")
        if self._all_drones is None or len(self._all_drones) == 0:
            raise Exception("No drones in simulation")
        if self._park_bounds is None:
            raise Exception("Park bounds is not set")
        if not self._random_trash_generation_on and not self._persons_on:
            raise Exception("No trash generation methods set on")

    def _set_drone_ids(self, sim_model):
        for index, drone in enumerate(sim_model.all_drones):
            drone.set_id(index)

    def _set_collector_ids(self, sim_model):
        for index, collector in enumerate(sim_model.all_collectors):
            collector.set_id(index)

    def _set_charger_ids(self, sim_model):
        for index, charger in enumerate(sim_model.all_rechargers):
            charger.set_id(index)

    def _set_potential_fields_is_active(self, sim_model):
        for drone in sim_model.all_drones:
            if drone.potential_fields_on:
                return True
        return False

    def _set_sim_model_parameters(self, sim_model):
        sim_model.random_trash_generation_on = self._random_trash_generation_on
        sim_model.trash_spawning_rate = self._trash_spawning_rate

        sim_model.persons_on = self._persons_on
        sim_model.person_params = self._person_params
        sim_model.person_spawning_rate = self._person_spawning_rate

        sim_model.all_collectors = self._all_collectors
        sim_model.all_drones = self._all_drones
        sim_model.all_rechargers = self._all_rechargers
        sim_model.collector_coords = [collector.position for collector in sim_model.all_collectors]
        sim_model.charger_coords = [charger.position for charger in sim_model.all_rechargers]

        sim_model.park = Park(self._park_bounds, self._persons_on)
        sim_model.potential_fields_on = self._set_potential_fields_is_active(sim_model)
```

sim_data_logger.py

```
import time
from math import floor

import numpy as np
import matplotlib.pyplot as plt

from parkcleanup.parkcleanup.model.agents.drone_state_type import DroneStateType

class SimDataLogger():
    def __init__(self, trash_heatmap_disc, search_heatmap_disc, experiment_mode, hm_at_every_time_step=
     True):
        # Experiment mode minimizes RAM by only computing running averages,
        # if false it will record all information needed to plot an experiment
        self.experiment_mode = experiment_mode
        self.hm_at_every_time_step = hm_at_every_time_step
        self._initialize_drone_metrics(search_heatmap_disc)
        self._initialize_trash_metrics(trash_heatmap_disc)

    def _initialize_drone_metrics(self, search_heatmap_disc):
        self.num_time_visited_hm = np.zeros((search_heatmap_disc, search_heatmap_disc))
        self.time_last_searched_hm = np.zeros((search_heatmap_disc, search_heatmap_disc))
        self.running_sum_total = np.zeros((search_heatmap_disc, search_heatmap_disc))
        self.running_sum_squared_total = np.zeros((search_heatmap_disc, search_heatmap_disc))
        self.search_hm_disc = search_heatmap_disc
        self.all_drone_heat_map = []
        self.all_max_hm = []
        self.all_mean_hm = []
        self.all_std_dev_hm = []
        self.total_time_spent_searching = 0
        self.total_time_spent_searching_sq = 0
        self.total_time_spent_collecting = 0
        self.total_time_spent_collecting_sq = 0
        self.drones_with_depleted_energy = set([])
        self.drones_with_depleted_energy_times = []
        self.num_drones_collecting = []
        self.num_drones_searching = []

    def _initialize_trash_metrics(self, trash_heatmap_disc):
        self.trash_heatmap_disc = trash_heatmap_disc
        # Record how many trash in the sim at each time step
        self.num_trash_each_time_step = []
        # Used for calculating the running average of how many trash in sim
        self.total_trash_counting_duplicates = 0
        self.running_avg_num_trash_each_time_step = []

        self.total_number_of_trash_collected = 0
        self.total_collected_trash_times = 0

        self.all_trash_info = []
        # Used for calculating stats related to average of
        # (sum of times of trash in time step i)/(Number of trash out in time step i)
        self.trash_time_at_each_time_step = []

        self.running_avg_of_avg_time_left_out_at_each_time_step = []
        self.sum_of_trash_times = 0
        self.sum_of_squared_trash_times = 0

        self.longest_time_left_out = 0
        self.longest_curr_trash_left_out = []

        self.times_left_out_heat_map = np.zeros((trash_heatmap_disc, trash_heatmap_disc))
        self.num_trash_collected_heat_map = np.zeros((trash_heatmap_disc, trash_heatmap_disc))
        if not self.experiment_mode:
            self.avg_heat_map = np.zeros((trash_heatmap_disc, trash_heatmap_disc))
            self.all_avg_trash_hm = []

        self.additional_trash_at_end = 0
        self.additional_times_at_end = 0
        self.additional_times_at_end_sq = 0

    def _initialize_visualization_metrics(self):
        self.drone_history = [None]*self.num_time_steps
        self.drone_battery_life = [None]*self.num_time_steps
        self.active_drones_history = [None]*self.num_time_steps
        self.searching_drones_history = [None]*self.num_time_steps
        self.trash_history = [None]*self.num_time_steps
        self.longest_trash_index = [None]*self.num_time_steps

    def update_initial_information(self, park_sim):
        bounds = park_sim.sim_model.park.bounds
        tdr = park_sim.sim_model.all_drones[0].trash_detection_radius
        self.random_seed = park_sim.random_seed
        self.num_time_steps = park_sim.num_time_steps
        self.bounds = bounds
        self.tdr = tdr
        self.num_drones = len(park_sim.sim_model.all_drones)
        self.drone_hm_lookup_table = self._initialize_discretized_drone_search_radius_lookup_table(bounds
    , self.search_hm_disc, tdr)
        self.collector_positions = [collector.position for collector in park_sim.sim_model.all_collectors
    ]
        self.charger_positions = [charger.position for charger in park_sim.sim_model.all_rechargers]
        if not self.experiment_mode:
            self._initialize_visualization_metrics()

    def update(self, index, sim_model):
        self._update_drone_information(sim_model.all_drones, index)
        self._update_trash_information(sim_model.all_trash,
                                       sim_model.trash_ids,
                                       sim_model.start_times,
                                       sim_model.times_left_out,
                                       sim_model.times_left_out_positions,
                                       index)

    def update_final_information(self, sim):
        self.total_number_of_trash = sim.trash_id_counter
        times_left_out = [trash.time_left_out for trash in sim.sim_model.all_trash]
        self.additional_trash_at_end = len(times_left_out)
        if len(times_left_out) != 0:
            self.additional_times_at_end += sum(times_left_out)
            self.additional_times_at_end_sq += (
                sum([trash.time_left_out**2 for trash in sim.sim_model.all_trash]))
        for trash in sim.sim_model.all_trash:
            # Use negative one to denote the trash was not picked up at the end
            self.all_trash_info.append([trash.id,
                                       trash.start_time,
                                       -1,
                                       trash.position[0],
                                       trash.position[1]])

    def _update_trash_information(self, all_trash, trash_ids, start_times,
                                  collected_trash_times, collected_positions, index):
        # Metrics related to number of trash left out
        num_trash_rn = len(all_trash)
        self.num_trash_each_time_step.append(num_trash_rn)
        # Running avg num trash
        self.total_trash_counting_duplicates += num_trash_rn
        self.running_avg_num_trash_each_time_step.append(self.total_trash_counting_duplicates/(index+1))

        # Stats on collected trash
        if len(collected_trash_times) != 0:
            self.total_collected_trash_times += sum(collected_trash_times)
            self.total_number_of_trash_collected += len(collected_trash_times)
            for trash_id, start_time, collected_position, collected_time in zip(
                                                          trash_ids,
                                                          start_times,
                                                          collected_positions,
                                                          collected_trash_times):
                self._update_trash_hm(collected_position, collected_time)
                self.all_trash_info.append([trash_id,
                                           start_time,
                                           collected_time,
                                            collected_position[0],
                                            collected_position[1]
                                            ])
        if not self.experiment_mode:
            if self.hm_at_every_time_step:
                self.all_avg_trash_hm.append(np.copy(self.avg_heat_map))

        # Stats on current trash in simulation
        times_left_out = [trash.time_left_out for trash in all_trash]
        if len(times_left_out) != 0:
            self.longest_curr_trash_left_out.append(max(times_left_out))
            sum_trash_times = sum(times_left_out)
            self.trash_time_at_each_time_step.append(sum_trash_times)
            self.sum_of_trash_times += sum_trash_times
            self.sum_of_squared_trash_times += sum([trash.time_left_out**2 for trash in all_trash])
        else:
            self.longest_curr_trash_left_out.append(0)
            self.trash_time_at_each_time_step.append(0)

        if self.total_trash_counting_duplicates == 0:
            self.running_avg_of_avg_time_left_out_at_each_time_step.append(0)
        else:
            self.running_avg_of_avg_time_left_out_at_each_time_step.append(
                self.sum_of_trash_times/(self.total_trash_counting_duplicates))
        if not self.experiment_mode:
            self.trash_history[index] = [trash.position for trash in all_trash]
            if self.longest_curr_trash_left_out[-1] == 0:
                # Mark with negative one when no trash is in the sim
                # so plotter handles accordingly
                self.longest_trash_index[index] = -1
            else:
                self.longest_trash_index[index] = times_left_out.index(self.longest_curr_trash_left_out
 [-1])

    def _update_trash_hm(self, position, time_left_out):
        bounds = self.bounds
        discretization = self.trash_heatmap_disc
        x_grid_position = int(floor(position[0]/bounds*discretization))
        y_grid_position = int(floor(position[1]/bounds*discretization))
        self.times_left_out_heat_map[x_grid_position][y_grid_position] += time_left_out
        self.num_trash_collected_heat_map[x_grid_position][y_grid_position] += 1
        if not self.experiment_mode:
            self.avg_heat_map[x_grid_position][y_grid_position] = (
                self.times_left_out_heat_map[x_grid_position][y_grid_position] /
                self.num_trash_collected_heat_map[x_grid_position][y_grid_position])

    def get_max_trash_indices(self):
        return self.longest_trash_index
    def _get_x_for_plotting(self):
        return list(range(self.num_time_steps))

    def get_total_trash_time_per_time_step_data(self):
        return self._get_x_for_plotting(), self.trash_time_at_each_time_step

    def get_trash_per_time_step_data(self):
        return self._get_x_for_plotting(), self.num_trash_each_time_step

    def get_running_avg_num_trash_per_timestep_data(self):
        return self._get_x_for_plotting(), self.running_avg_num_trash_each_time_step

    def max_trash_left_out_each_time_step_data(self):
        return self._get_x_for_plotting(), self.longest_curr_trash_left_out

    def avg_time_trash_left_out_in_each_time_step_data(self):
        return self._get_x_for_plotting(), self.running_avg_of_avg_time_left_out_at_each_time_step

    def get_avg_time_trash_left_out(self):
        if self.total_number_of_trash_collected+self.additional_trash_at_end ==0:
            return 0
        else:
            return (
            (self.total_collected_trash_times+self.additional_times_at_end)
            /(self.total_number_of_trash_collected+self.additional_trash_at_end)
            )

    def get_avg_time_trash_collected(self):
        if self.total_number_of_trash_collected == 0:
            return 0
        else:
            return self.total_collected_trash_times/self.total_number_of_trash_collected



    def get_std_dev_time_trash_left_out(self):
        return self._std_dev(self.sum_of_trash_times+self.additional_times_at_end,
                             self.sum_of_squared_trash_times+self.additional_times_at_end_sq,
                             self.total_number_of_trash)

    def get_max_time_any_trash_left_out(self):
        return max(self.longest_curr_trash_left_out)

    def get_avg_num_trash_in_sim(self):
        return self.running_avg_num_trash_each_time_step[-1]

    def get_max_num_trash_in_sim_any_time(self):
        return max(self.num_trash_each_time_step)

    def get_std_dev_num_trash_in_sim(self):
        return np.std(self.num_trash_each_time_step)

    def get_num_trash_collected_heat_map(self):
        return self.num_trash_collected_heat_map

    def get_avg_collected_time_heat_map(self):
        return self.times_left_out_heat_map/self.num_trash_collected_heat_map

    def get_total_trash_picked_up(self):
        return self.total_number_of_trash_collected

    def get_total_number_of_unique_trash_in_sim(self):
        return self.total_number_of_trash


    def _update_drone_information(self, all_drones, index):
        self.time_last_searched_hm += 1
        search_time = 0
        collecting_time = 0
        num_drones_collecting = 0
        num_drones_searching = 0
        if not self.experiment_mode:
            drone_positions = []
            drone_battery_life = []
        for drone in all_drones:
            if (drone._state_type == DroneStateType.GO_TO_TRASH
                or drone._state_type == DroneStateType.SEARCH_FOR_TRASH):
                self._update_drone_search_hms(drone.position)
            if drone._state_type in (DroneStateType.PICK_UP_TRASH,
                                     DroneStateType.DROP_OFF_TRASH,
                                     DroneStateType.GO_TO_COLLECTOR,
                                     DroneStateType.GO_TO_TRASH):
                collecting_time += 1
                num_drones_collecting += 1
            elif drone._state_type == DroneStateType.SEARCH_FOR_TRASH:
                search_time += 1
                num_drones_searching += 1
            elif drone._state_type == DroneStateType.OUT_OF_ENERGY:
                if drone.id not in self.drones_with_depleted_energy:
                    self.drones_with_depleted_energy.add(drone.id)
                    self.drones_with_depleted_energy_times.append(index)
            if not self.experiment_mode:
                drone_positions.append(drone.position)
                drone_battery_life.append(drone.battery_life)
        if not self.experiment_mode:
            self.active_drones_history[index] = num_drones_collecting
            self.searching_drones_history[index] = num_drones_searching
            self.drone_history[index] = drone_positions
            self.drone_battery_life[index] = drone_battery_life

        self.running_sum_total += self.time_last_searched_hm
        self.running_sum_squared_total += self.time_last_searched_hm**2
        if not self.experiment_mode:
            if self.hm_at_every_time_step:
                self.all_drone_heat_map.append(np.copy(self.time_last_searched_hm))
        self.total_time_spent_searching += search_time
        self.total_time_spent_searching_sq += search_time**2
        self.total_time_spent_collecting += collecting_time
        self.num_drones_searching.append(num_drones_searching)
        self.num_drones_collecting.append(num_drones_collecting)

        curr_max = np.max(self.time_last_searched_hm).item(0)
        self.all_max_hm.append(curr_max)
        curr_mean = np.mean(self.time_last_searched_hm).item(0)
        self.all_mean_hm.append(curr_mean)
        curr_std_dev = np.std(self.time_last_searched_hm).item(0)
        self.all_std_dev_hm.append(curr_std_dev)

    def get_num_drones_ran_out_of_batteries(self):
        return len(self.drones_with_depleted_energy)

    def get_avg_time_spent_searching_per_drone(self):
        return self.total_time_spent_searching/self.num_drones

    def get_std_dev_time_spent_searching_per_drone(self):
        return self._std_dev(self.total_time_spent_searching,
                             self.total_time_spent_searching_sq,
                             self.num_drones)

    def get_avg_time_spent_collecting_per_drone(self):
        return self.total_time_spent_collecting/self.num_drones

    def get_std_dev_time_spent_collecting_per_drone(self):
        return self._std_dev(self.total_time_spent_collecting,
                             self.total_time_spent_collecting_sq,
                             self.num_drones)

    def _update_drone_search_hms(self, position):
        bounds = self.bounds
        discretization = self.search_hm_disc
        x_grid_position = int(floor(position[0]/bounds*discretization))
        y_grid_position = int(floor(position[1]/bounds*discretization))
        # If the UAV by chance goes outside the park, there will be no entry in the
        # lookup table, and so the grid cells seen from there must be calculated again
```

```python
        if x_grid_position < 0 or x_grid_position >= discretization or y_grid_position < 0 or y_grid_position >= discretization:
            cells_drone_can_see = self._get_indices_of_cells_drone_can_see_inside_map(
                self._cell_indices_drone_can_see_from_center,
                x_grid_position, y_grid_position,
                discretization)
            if len(self.time_last_searched_hm[cells_drone_can_see[:,0], cells_drone_can_see[:,1]]) == 0:
                return
        else:
            cells_drone_can_see = self.drone_hm_lookup_table[x_grid_position][y_grid_position]
        self.time_last_searched_hm[cells_drone_can_see[:,0], cells_drone_can_see[:,1]] = 0
        self.num_time_visited_hm[cells_drone_can_see[:,0], cells_drone_can_see[:,1]] += 1

    def _initialize_discretized_drone_search_radius_lookup_table(self, bounds, discretization, tdr, print_checkpoint=False):
        self._cell_indices_drone_can_see_from_center = self._get_cell_indices_drone_can_see_from_center(bounds, discretization, tdr)
        if print_checkpoint:
            print("Start drone heat map preallocation")
            start = time.time()
        # Create lookup table for all the cells that a drone can see from each grid cell
        # Depending on its detection radius
        lookup_table = []
        for i in range(discretization):
            row = []
            for j in range(discretization):
                cells_drone_can_see = self._get_indices_of_cells_drone_can_see_inside_map(
                    self._cell_indices_drone_can_see_from_center,
                    i, j,
                    discretization)
                row.append(cells_drone_can_see)
            lookup_table.append(row)
        if print_checkpoint:
            end = time.time()
            print("Time: {}".format(str(end-start)))
        return lookup_table

    def _get_indices_of_cells_drone_can_see_inside_map(self, center_circle, i, j, discretization):
        cells_drone_can_see = center_circle + [i, j]
        # Only include the cells that are inside the map
        indices_to_take = np.argwhere(np.all(np.logical_and(cells_drone_can_see < discretization, cells_drone_can_see >= 0), axis=1)).flatten()
        cells_drone_can_see = cells_drone_can_see[indices_to_take]
        return cells_drone_can_see

    def _get_cell_indices_drone_can_see_from_center(self, bounds, discretization, tdr):
        # Convert float radius to radius in number of cells
        map_len = bounds
        cell_len = map_len/discretization
        cell_radius_float = tdr/cell_len
        cell_radius = int(round(cell_radius_float))+1

        # Create indices of every cell in the park
        m, n = discretization, discretization
        xs = np.arange(m)
        ys = np.arange(n)
        x = xs - m/2
        y = ys - n/2
        X, Y = np.meshgrid(x, y)
        # Find cells that are within cell radius**2 and save indices
        center_circle = np.argwhere((X**2 + Y**2) <= cell_radius**2)
        center_circle[:,0] = center_circle[:,0] - m/2
        center_circle[:,1] = center_circle[:,1] - n/2
        center_circle = np.unique(center_circle, axis=0)
        return center_circle

    def get_num_times_visited_hm(self):
        return self.num_time_visited_hm

    def get_average_time_trash_in_cell_hms(self):
        return self.all_avg_trash_hm

    def get_all_last_search_heat_map(self):
        return self.all_drone_heat_map

    def get_average_heat_map(self):
        return self.running_sum_total/self.num_time_steps

    def get_std_deviation_heat_map(self):
        #https://en.wikipedia.org/wiki/Algorithms_for_calculating_variance
        # sqrt of naive variance with bessels correction
        rs = self.running_sum_total
        rs_sq = self.running_sum_squared_total
        nts = self.num_time_steps
        return self._std_dev(rs, rs_sq, nts)

    def _std_dev(self, sum_, sum_sq, N):
        if N==1:
            return -1
        else:
            return np.sqrt((sum_sq - sum_**2/N)/N)

    def _plot_heat_map(self):
        heat_map = self.get_std_deviation_heat_map()
        fig, ax = plt.subplots()
        bounds = self.bounds
        extent = (0,bounds,0,bounds)
        hm = ax.imshow(heat_map.T, vmin=0, vmax=np.max(heat_map), interpolation='nearest', origin='lower'
    , extent=extent)
        # ax.set_title(title)
        plt.colorbar(hm)
        plt.show()
        # plt.savefig(PathManager.plot_save_output_path(self._doe_name, title, index))
        # plt.close(fig=fig)
```

park.py

```
from random import choice

class Park(object):
    def __init__(self, bounds, nodes_on):
        self.bounds = bounds
        self.nodes_on = nodes_on
        if nodes_on:
            x1 = 0
            x2 = 0.1*bounds
            x3 = 0.35*bounds
            x4 = 0.5*bounds
            x5 = 0.65*bounds
            x6 = 0.9*bounds
            x7 = 1.0*bounds
            y1 = 0
            y2 = 0.1*bounds
            y3 = 0.35*bounds
            y4 = 0.5*bounds
            y5 = 0.65*bounds
            y6 = 0.9*bounds
            y7 = 1.0*bounds
            A = Node([x1,y7],None,True)
            B = Node([x4,y7],None,True)
            C = Node([x7,y7],None,True)
            D = Node([x2,y6],None,False)
            E = Node([x4,y6],None,False)
            F = Node([x6,y6],None,False)
            G = Node([x3,y5],None,False)
            H = Node([x4,y5],None,False)
            I = Node([x5,y5],None,False)
            J = Node([x1,y4],None,True)
            K = Node([x2,y4],None,False)
            L = Node([x3,y4],None,False)
            M = Node([x4,y4],None,False)
            N = Node([x5,y4],None,False)
            O = Node([x6,y4],None,False)
            P = Node([x7,y4],None,True)
            Q = Node([x3,y3],None,False)
            R = Node([x4,y3],None,False)
            S = Node([x5,y3],None,False)
            T = Node([x2,y2],None,False)
            U = Node([x4,y2],None,False)
            V = Node([x6,y2],None,False)
            W = Node([x1,y1],None,True)
            X = Node([x4,y1],None,True)
            Y = Node([x7,y1],None,True)
            A.children = [D]
            B.children = [E]
            C.children = [F]
            D.children = [E,G,K,A]
            E.children = [D,F,H,B]
            F.children = [E,O,I,C]
            G.children = [D,L,H,M]
            H.children = [E,G,I,M]
            I.children = [F,H,M,N]
            J.children = [K]
            K.children = [L,D,T,J]
            L.children = [G,M,Q,K]
            M.children = [G,H,I,L,N,Q,R,S]
            N.children = [I,M,S,O]
            O.children = [F,N,V,P]
            P.children = [O]
            Q.children = [L,M,R,T]
            R.children = [M,Q,S,U]
            S.children = [N,M,R,V]
            T.children = [K,Q,U,W]
            U.children = [R,T,V,X]
            V.children = [S,O,U,Y]
            W.children = [T]
            X.children = [U]
            Y.children = [V]
            A.value = "A"
            B.value = "B"
            C.value = "C"
            D.value = "D"
            E.value = "E"
            F.value = "F"
            G.value = "G"
            H.value = "H"
            I.value = "I"
            J.value = "J"
            K.value = "K"
            L.value = "L"
            M.value = "M"
            N.value = "N"
            O.value = "O"
```

```
            P.value = "P"
            Q.value = "Q"
            R.value = "R"
            S.value = "S"
            T.value = "T"
            U.value = "U"
            V.value = "V"
            W.value = "W"
            X.value = "X"
            Y.value = "Y"
            A.value = "A"
            B.value = "B"
            C.value = "C"
            D.value = "D"
            E.value = "E"
            F.value = "F"
            G.value = "G"
            H.value = "H"
            I.value = "I"
            J.value = "J"
            K.value = "K"
            L.value = "L"
            M.value = "M"
            N.value = "N"
            O.value = "O"
            P.value = "P"
            Q.value = "Q"
            R.value = "R"
            S.value = "S"
            T.value = "T"
            U.value = "U"
            V.value = "V"
            W.value = "W"
            X.value = "X"
            Y.value = "Y"
            self.nodes = [A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y]
            self.exit_nodes = [A,B,C,P,W,X,Y,J]
            for node in self.nodes:
                node.visited = False

class Node(object):
    def __init__(self, coordinates, children, entrance_node):
        self.coordinates = coordinates
        self.children = children
        self.entrance_node = entrance_node

    def get_next_destination(self):
        return choice(self.children)
```

charge_station.py

```python
from parkcleanup.parkcleanup.model.objectives.objective import Objective
from parkcleanup.parkcleanup.model.objectives.objective_strings import CHARGER

ID = "ID"

class ChargeStation(Objective):
    def __init__(self, position, init_from_json=False):
        super().__init__(position)
        self.id = None

    def set_id(self, id_):
        self.id = id_

    @staticmethod
    def get_object_string_value():
        return CHARGER

    def export_to_json(self):
        values = super().export_to_json(ChargeStation.get_object_string_value())
        values[ID] = self.id
        return values

    def import_from_json(self, values):
        super().import_from_json(values)
        self.id = values[ID]
```

collector.py

```python
from parkcleanup.parkcleanup.model.objectives.objective import Objective
from parkcleanup.parkcleanup.model.objectives.objective_strings import COLLECTOR

ID = "ID"
TRASH_INSIDE = "Trash inside"

class Collector(Objective):
    def __init__(self, position, init_from_json=False):
        super().__init__(position)
        self.trash_inside = 0
        self.id = None

    def set_id(self, id_):
        self.id = id_

    @staticmethod
    def get_object_string_value():
        return COLLECTOR

    def export_to_json(self):
        values = super().export_to_json(Collector.get_object_string_value())
        values[TRASH_INSIDE] = self.trash_inside
        values[ID] = self.id
        return values

    def import_from_json(self, values):
        super().import_from_json(values)
        self.trash_inside = values[TRASH_INSIDE]
        self.id = values[ID]
        return self
```

trash.py

```
from parkcleanup.parkcleanup.model.objectives.objective import Objective
from parkcleanup.parkcleanup.model.objectives.objective_strings import TRASH

TIME_LEFT_OUT = "Time left out"


class Trash(Objective):
    def __init__(self, position, time_to_complete_dropoff, start_time, id, init_from_json=False):
        super().__init__(position)
        self.id = id
        self.start_time = start_time
        self.time_left_out = 0
        self.distances_to_drones = []
        self.time_to_complete_dropoff = time_to_complete_dropoff

    @staticmethod
    def get_object_string_value():
        return TRASH

    def export_to_json(self):
        values = super().export_to_json(Trash.get_object_string_value())
        values[TIME_LEFT_OUT] = self.time_left_out
        return values

    def import_from_json(self, values):
        super().import_from_json(values)
        self.time_left_out = values[TIME_LEFT_OUT]
        return self
```

location.py

```
from parkcleanup.parkcleanup.model.objectives.objective import Objective
from parkcleanup.parkcleanup.model.objectives.objective_strings import LOCATION


```

```python
class Location(Objective):
    def __init__(self, position, init_from_json=False):
        super().__init__(position)

    @staticmethod
    def get_object_string_value():
        return LOCATION

    def export_to_json(self):
        return super().export_to_json(Location.get_object_string_value())

    def import_from_json(self, values):
        super().import_from_json(values)
```

objective.py

```python
import abc

TYPE = "Objective Type"
POSITION = "Position"

class Objective(abc.ABC):
    def __init__(self, position, init_from_json=False):
        self.position = position

    def export_to_json(self, objective_type):
        values = {}
        values[TYPE] = objective_type
        values[POSITION] = self.position
        return values

    def import_from_json(self, values):
        self.position = values[POSITION]
        return self
```

objective_strings.py

```python
TRASH = "Trash"
COLLECTOR = "Collector"
CHARGER = "Charger"
LOCATION = "Location"
```

A.1.1 UAV Code

drone.py

```python
from enum import Enum
import abc
```

```
from math import sqrt
from random import randint
from random import random

from scipy.spatial import distance_matrix

from parkcleanup.parkcleanup.model.agents.movable import Movable
from parkcleanup.parkcleanup.model.agents.drone_state_type import DroneStateType
from parkcleanup.parkcleanup.model.objectives.trash import Trash
from parkcleanup.parkcleanup.model.objectives.location import Location
from parkcleanup.parkcleanup.model.objectives.objective import Objective
from parkcleanup.parkcleanup.model.drone_strategies.drone_search_strategies import _PatrolSearch,
    _RandomSearch, _RandomBounceSearch
from parkcleanup.parkcleanup.model.drone_strategies.drone_path_planning_strategies import _DirectRoute,
    _PotentialFields
from parkcleanup.parkcleanup.tools.helper import distance

class Drone(Movable):
    def __init__(self, bounds):
        # Use DroneBuilder for initialization
        self.position = None
        self.direction = None
        self.speed = None

        self.potential_fields_on = None
        self.avoidance_distance = None
        self.repulse_radius = None
        self.attract_scale = None

        self.found_distance = None
        self.patrol_coordinates = None
        self.group_index = None
        self.bounds = bounds

        self.id = None
        self.trash_detection_radius = None
        self.emergency_recharge_level = None
        self.set_out_for_seen_trash_while_charging = None
        self.return_to_charge_from_patrolling = None

        self.fly_time = None # seconds
        self.recharge_time = None
        self.can_communicate_objective = False
        self.cant_see_trash_sometimes = False

        self.trash_pickup_delay = None
        self.trash_dropoff_delay = None
        self.wait_to_start = None
```

```python

        self.objective = None
        self._state_type = None
        # Start drone with a full charge
        self.battery_life = 1 # from 0 to 1
        self.has_trash = False
        self.is_on_lookout_for_trash = True
        self.trash_held = []
        self.poly_of_area = None
        self.start_waypoint = None

        self._state = None
        self._state_dict = self._create_state_dict()

    def set_id(self, id):
        self.id = id

    def update(self, sim_model):
        state = self._state
        state.update_energy(self, sim_model)
        if self.battery_life < 0:
            self._set_state(DroneStateType.OUT_OF_ENERGY, sim_model)
        else:
            state.update_objective(self, sim_model)
            if self.objective is not None:
                self._path_planning_strategy.update_direction(self, sim_model)
                self._update_coordinates()

    def set_path_planning_method(self, path_planning_type):
        if not isinstance(path_planning_type, PathPlanningType):
            raise TypeError("path_planning_type must be of type PathPlanningType")
        if path_planning_type == PathPlanningType.DIRECT_ROUTE:
            self._path_planning_strategy = _DirectRoute()
        elif path_planning_type == PathPlanningType.POTENTIAL_FIELDS:
            self._path_planning_strategy = _PotentialFields()
        else:
            raise ValueError("Path planning behavior not implemented")

    def set_search_method(self, search_type):
        if not isinstance(search_type, SearchType):
            raise TypeError("search_type must be of type SearchType")
        if search_type == SearchType.RANDOM_SEARCH:
            self._search_strategy = _RandomSearch()
        elif search_type == SearchType.PATROL:
            self._search_strategy = _PatrolSearch(self.patrol_coordinates, closest_waypoint_on_resume=
    False)
        elif search_type == SearchType.RANDOM_BOUNCE:
            self._search_strategy = _RandomBounceSearch(self.poly_of_area, self.bounds)
        else:
            raise ValueError("Search behavior not implemented")

    def _set_state(self, state_type, sim_model):
        if not isinstance(state_type, DroneStateType):
            raise TypeError("Objective type must be of type ObjectiveType")
        self._state_type = state_type
        self._state = self._state_dict[state_type]
        self._state.initialize(self, sim_model)

    def _check_for_trash_to_pick_up(self, sim_model):
        if self.is_on_lookout_for_trash and sim_model.there_is_trash_in_model():
            all_trash_detected = self._look_for_trash(sim_model)
            if len(all_trash_detected) != 0:
                trash = self._decide_on_trash_to_pick_up(all_trash_detected)
                return True, trash
        return False, None

    def _look_for_trash(self, sim_model):
        trash_in_range = [trash for trash in sim_model.all_trash if trash.distances_to_drones[self.id] <
    self.trash_detection_radius]
        if self.cant_see_trash_sometimes:
            trash_detected = self._detect_trash(trash_in_range)
        else:
            trash_detected = trash_in_range
        return trash_detected

    def _detect_trash(self, trash_in_range):
        # TODO include detection criteria
        return trash_in_range

    def _decide_on_trash_to_pick_up(self, all_trash_detected):
        # Drone decides to travel to the closest trash
        closest_trash = all_trash_detected[0]
        for trash in all_trash_detected:
            if closest_trash.distances_to_drones[self.id] > trash.distances_to_drones[self.id]:
                closest_trash = trash
        return closest_trash

    def _reached_objective(self):
        return self.distance(self.position, self.objective.position) < self.found_distance

    def _decrease_energy(self):
        self._decrease_energy_linearly()

    def _decrease_energy_linearly(self):
        self.battery_life -= 1/self.fly_time

    def _increase_energy(self):
        self.battery_life += 1/self.recharge_time

    def _set_position_as_objective_position(self):
        self.position = self.objective.position

    def _create_state_dict(self):
        return {
            DroneStateType.GO_TO_TRASH: GoToTrashState(self),
            DroneStateType.GO_TO_COLLECTOR: GoToCollectorState(self),
            DroneStateType.SEARCH_FOR_TRASH: SearchForTrashState(self),
            DroneStateType.GO_TO_CHARGER: GoToChargerState(self),
            DroneStateType.RECHARGE: RechargeState(self),
            DroneStateType.DROP_OFF_TRASH: DropOffTrashState(self),
            DroneStateType.PICK_UP_TRASH: PickUpTrashState(self),
            DroneStateType.OUT_OF_ENERGY: OutOfEnergyState(self),
            DroneStateType.WAIT_TO_START: WaitToStartState(self),
            DroneStateType.LAND_ON_CHARGER: LandOnChargerState(self),
            DroneStateType.TAKE_OFF: TakeOffState(self)
        }

class SearchType(Enum):
    RANDOM_SEARCH = "Random Search"
    PATROL = "Patrolling"
    RANDOM_BOUNCE = "Random Bounce"

class PathPlanningType(Enum):
    DIRECT_ROUTE = "Direct Route"
    POTENTIAL_FIELDS = "Potential Fields"

class DroneObjectiveState(metaclass=abc.ABCMeta):
    def __init__(self, drone):
        self._drone = drone

    def initialize(self, drone, sim_model):
        pass

    def update_energy(self, drone, sim_model):
        pass

    def update_objective(self, drone, sim_model):
        pass

    def _decide_to_get_trash(self, drone, sim_model, trash):
        time_to_cross_park_thrice = (sim_model.park.bounds*sqrt(2)*3)/drone.speed
        if drone.battery_life*drone.fly_time < time_to_cross_park_thrice:
            distance_to_trash = distance(trash.position, drone.position)
            time_to_drop_off_trash = (distance_to_trash+trash.time_to_complete_dropoff)/drone.speed
            if time_to_drop_off_trash < drone.battery_life*drone.fly_time:
                drone.objective = trash
                drone._set_state(DroneStateType.GO_TO_TRASH, sim_model)
                return True
        else:
            drone.objective = trash
            drone._set_state(DroneStateType.GO_TO_TRASH, sim_model)
            return True
        return False

class WaitToStartState(DroneObjectiveState):
    def __init__(self, drone):
        super().__init__(drone)

    def initialize(self, drone, sim_model):
        self._countdown = drone.wait_to_start

    def update_energy(self, drone, sim_model):
        pass

    def update_objective(self, drone, sim_model):
        self._countdown -= 1
        if self._countdown <= 0:
            drone._set_state(DroneStateType.SEARCH_FOR_TRASH, sim_model)

class GoToTrashState(DroneObjectiveState):
    def __init__(self, drone):
        super().__init__(drone)

    def initialize(self, drone, sim_model):
        pass

    def update_energy(self, drone, sim_model):
        drone._decrease_energy()

    def update_objective(self, drone, sim_model):
        if drone._reached_objective():
            # Pick up trash
            drone.has_trash = True
            trash_coord = drone.objective.position
            drone.trash_held = drone.objective
            drone._set_position_as_objective_position()
            # Make sure the other drones don't go for picked up trash
            self._tell_other_drones_to_change_trash_obj(drone, sim_model, trash_coord)
            self._clean_up_trash(trash_coord, sim_model)
            drone._set_state(DroneStateType.PICK_UP_TRASH, sim_model)
        else:
            # Search for closer trash that may have appeared
            found_trash, trash = drone._check_for_trash_to_pick_up(sim_model)
            if found_trash:
                self._decide_to_get_trash(drone, sim_model, trash)

    def _time_to_recharge(self, drone):
        return drone.battery_life < drone.emergency_recharge_level

    def _tell_other_drones_to_change_trash_obj(self, drone, sim_model, trash_coord):
        for drone in sim_model.all_drones:
            if drone == self:
                pass
            elif isinstance(drone.objective, Trash) and drone.objective.position == trash_coord:
                drone.objective = None
                drone._set_state(DroneStateType.SEARCH_FOR_TRASH, sim_model)

    def _clean_up_trash(self, trash_coord, sim_model):
        trash_index = sim_model.trash_coords.index(trash_coord)
        for row in sim_model.drone_to_trash:
            del row[trash_index]
        del sim_model.trash_coords[trash_index]
        # Recording of any details about the trash cleanup should be done here
        sim_model.record_trash_pickup_event(sim_model.all_trash[trash_index])
        del sim_model.all_trash[trash_index]

class GoToCollectorState(DroneObjectiveState):
    def __init__(self, drone):
        super().__init__(drone)

    def initialize(self, drone, sim_model):
        collectors = [collector.position for collector in sim_model.all_collectors]
        distances_to_collectors = distance_matrix([drone.position], collectors).tolist()
        index_of_min_distance_recharger = distances_to_collectors[0].index(min(distances_to_collectors
    [0]))
        drone.objective = sim_model.all_collectors[index_of_min_distance_recharger]

    def update_energy(self, drone, sim_model):
        drone._decrease_energy()

    def update_objective(self, drone, sim_model):
        if drone._reached_objective():
            self._collect_trash(drone, sim_model)
            drone._set_position_as_objective_position()
            drone._set_state(DroneStateType.DROP_OFF_TRASH, sim_model)

    def _collect_trash(self, drone, sim_model):
        collector_coords = sim_model.collector_coords
        trash = drone.trash_held
        drone.has_trash = False
        drone.trash_held = None
        collector_coord = drone.objective.position
        collector_index = collector_coords.index(collector_coord)
        collector = sim_model.all_collectors[collector_index]

class SearchForTrashState(DroneObjectiveState):
    def __init__(self, drone):
        super().__init__(drone)

    def initialize(self, drone, sim_model):
        drone._search_strategy.update_strategy_on_state_change(drone, sim_model)

    def update_energy(self, drone, sim_model):
        drone._decrease_energy()

    def update_objective(self, drone, sim_model):
        drone._search_strategy.search_update_method(drone, sim_model)
        found_trash, trash = drone._check_for_trash_to_pick_up(sim_model)
        if found_trash:
            self._decide_to_get_trash(drone, sim_model, trash)
        if drone.battery_life*drone.fly_time < (sim_model.park.bounds*sqrt(2))/drone.speed:
            charger_dist = min(distance_matrix(sim_model.charger_coords, [drone.position]))
            # Have a little buffer
            if drone.battery_life*drone.fly_time < (charger_dist/drone.speed)+2:
                drone._set_state(DroneStateType.GO_TO_CHARGER, sim_model)

class GoToChargerState(DroneObjectiveState):
    def __init__(self, drone):
        super().__init__(drone)

    def initialize(self, drone, sim_model):
        rechargers = [charger.position for charger in sim_model.all_rechargers]
        distances_to_rechargers = distance_matrix([drone.position], rechargers).tolist()
        index_of_min_distance_recharger = distances_to_rechargers[0].index(min(distances_to_rechargers
    [0]))
        drone.objective = sim_model.all_rechargers[index_of_min_distance_recharger]

    def update_energy(self, drone, sim_model):
        drone._decrease_energy()

    def update_objective(self, drone, sim_model):
        if drone._reached_objective():
            drone._set_position_as_objective_position()
            drone._set_state(DroneStateType.RECHARGE, sim_model)

class LandOnChargerState(DroneObjectiveState):
    def __init__(self, drone):
        super().__init__(drone)
```

```python

    def initialize(self, drone, sim_model):
        self._countdown = 1

    def update_energy(self, drone, sim_model):
        drone._decrease_energy()

    def update_objective(self, drone, sim_model):
        self._countdown -= 1
        if self._countdown < 0:
            drone._set_state(DroneStateType.RECHARGE, sim_model)

class RechargeState(DroneObjectiveState):
    def __init__(self, drone):
        super().__init__(drone)

    def initialize(self, drone, sim_model):
        self._charger_drone_landed_on = drone.objective
        self._starting_time_step = sim_model.curr_time_step
        drone.objective = None

    def update_energy(self, drone, sim_model):
        drone._increase_energy()

    def update_objective(self, drone, sim_model):
        if drone.battery_life > drone.set_out_above_this:
            found_trash, trash = drone._check_for_trash_to_pick_up(sim_model)
            if found_trash:
                self._decide_to_get_trash(drone, sim_model, trash)
        elif drone.battery_life > 0.99:
            drone._set_state(DroneStateType.TAKE_OFF, sim_model)

class TakeOffState(DroneObjectiveState):
    def __init__(self, drone):
        super().__init__(drone)

    def initialize(self, drone, sim_model):
        self._countdown = 1

    def update_energy(self, drone, sim_model):
        drone._decrease_energy()

    def update_objective(self, drone, sim_model):
        self._countdown -= 1
        if self._countdown < 0:
            drone._set_state(DroneStateType.SEARCH_FOR_TRASH, sim_model)

class DropOffTrashState(DroneObjectiveState):
    def __init__(self, drone):
        super().__init__(drone)

    def initialize(self, drone, sim_model):
        drone.objective = None
        self._time_spent_dropping_off = 0
        self._max_time_to_drop_off = drone.trash_dropoff_delay

    def update_energy(self, drone, sim_model):
        drone._decrease_energy()

    def update_objective(self, drone, sim_model):
        self._time_spent_dropping_off += 1
        if self._time_spent_dropping_off == self._max_time_to_drop_off:
            found_trash, trash = drone._check_for_trash_to_pick_up(sim_model)
            if found_trash:
                going_to_trash = self._decide_to_get_trash(drone, sim_model, trash)
                if going_to_trash:
                    return
            drone._set_state(DroneStateType.SEARCH_FOR_TRASH, sim_model)

class PickUpTrashState(DroneObjectiveState):
    def __init__(self, drone):
        super().__init__(drone)

    def initialize(self, drone, sim_model):
        drone.objective = None
        self._time_spent_picking_up = 0
        self._max_time_to_pick_up = drone.trash_pickup_delay

    def update_energy(self, drone, sim_model):
        drone._decrease_energy()

    def update_objective(self, drone, sim_model):
        self._time_spent_picking_up += 1
        if self._time_spent_picking_up == self._max_time_to_pick_up:
            drone._set_state(DroneStateType.GO_TO_COLLECTOR, sim_model)

class OutOfEnergyState(DroneObjectiveState):
    def __init__(self, drone):
        super().__init__(drone)

    def initialize(self, drone, sim_model):
        drone.objective = None
        self._time_step_out_of_batteries = sim_model.curr_time_step

    def update_energy(self, drone, sim_model):
        pass
    def update_objective(self, drone, sim_model):
        pass
```

drone_builder.py

```
import random
from math import ceil, sqrt, floor
import copy

from parkcleanup.parkcleanup.tools.helper import random_position_in_bounds
from parkcleanup.parkcleanup.model.agents.drone import Drone
from parkcleanup.parkcleanup.model.agents.drone import SearchType
from parkcleanup.parkcleanup.model.agents.drone import PathPlanningType
from parkcleanup.parkcleanup.model.agents.drone_state_type import DroneStateType
from parkcleanup.parkcleanup.tools.coverage_path_generator.coverage_patterns import
     global_lawnmower_coords, partitioned_coords, get_partitions, get_square_poly

class DroneBuilder(object):
    def __init__(self, bounds):
        if bounds <= 0:
            raise ValueError("Bounds must be positive number")
        self._bounds = bounds

        self._potential_fields_on = None
        self._repulse_radius = None
        self._attract_scale = None
        self._avoidance_distance = None

        self._patrol_coordinates = None

        self._fly_time = None
        self._recharge_time = None

        self._direction = None
        self._speed = None
        self._position = None
        self._random_position = None

        self._num_drones = None
        self._search_method = None
        self._can_communicate_objective = None
        self._emergency_recharge_level = None
        self._set_out_for_seen_trash_while_charging = None
        self._return_to_charge_from_patrolling = None
        self._constant_trash_pickup_delay = None
        self._constant_trash_dropoff_delay = None
        self._trash_detection_radius = None
        self._starting_position_on_coordinates = False
        self._half_reverse_patrol = False
        self._wait_to_start = False
        self._index_to_wait_time = None
        self._group_index = None
        self._partitioned_lawnmower = False
        self._partitioned_polys = None
        self._global_pattern = False

    def set_constant_trash_pickup_delay(self, trash_pickup_delay):
        if not isinstance(trash_pickup_delay, int) and not trash_pickup_delay.is_integer():
            raise TypeError("Constant trash pickup delay must be int")
        if trash_pickup_delay < 0:
            raise ValueError("Constant trash pickup delay must be positive")
        self._constant_trash_pickup_delay = trash_pickup_delay
        return self

    def set_constant_trash_dropoff_delay(self, trash_dropoff_delay):
        if not isinstance(trash_dropoff_delay, int) and not trash_dropoff_delay.is_integer():
            raise TypeError("Constant trash dropoff delay must be int")
        if trash_dropoff_delay < 0:
            raise ValueError("Constant trash dropoff delay must be positive")
        self._constant_trash_dropoff_delay = trash_dropoff_delay
        return self

    def set_start_delay(self):
        self._wait_to_start = True
        return self

    def set_potential_fields_on(self, repulse_radius, attract_scale, avoidance_distance):
        self._potential_fields_on = True
        self._repulse_radius = repulse_radius
        self._attract_scale = attract_scale
        self._avoidance_distance = avoidance_distance
        return self
        # TODO check if input parameters are the right type

    def set_fly_time(self, fly_time_in_seconds):
        if not isinstance(fly_time_in_seconds, int) and not fly_time_in_seconds.is_integer():
            raise TypeError("Fly time must be an int")
        if fly_time_in_seconds <= 0:
            raise ValueError("Fly time must be positive")
        self._fly_time = fly_time_in_seconds
        return self

    def set_recharge_time(self, recharge_time_in_seconds):
        if not isinstance(recharge_time_in_seconds, int) and not recharge_time_in_seconds.is_integer():
            raise TypeError("Recharge time must be an int")
        if recharge_time_in_seconds <= 0:
            raise ValueError("Recharge time must be positive")
        self._recharge_time = recharge_time_in_seconds
        return self

    def set_search_method_patrol(self, patrol_coordinates, global_pattern=False):
        if not isinstance(patrol_coordinates, list):
            raise TypeError("Patrol coordinates must be a list of two value objects")
        for coordinates in patrol_coordinates:
            if len(coordinates) != 2:
                raise TypeError("Patrol coordinates must be a list of two value objects")
        self._patrol_coordinates = patrol_coordinates
        self._search_method = SearchType.PATROL
        self._global_pattern = True
        return self

    def set_search_method_global_lawnmower(self):
        if self._trash_detection_radius is None:
            raise Exception("Trash detection radius must be set before lawnmower search")
        patrol_coords = global_lawnmower_coords(self._bounds, self._trash_detection_radius, self._speed)
        return self.set_search_method_patrol(patrol_coords, global_pattern=True)

    def set_search_method_partitioned_lawnmower(self):
        self._partitioned_lawnmower = True
        self._search_method = SearchType.PATROL
        return self

    def set_search_method_partitioned_random_bounce(self):
        self._search_method = SearchType.RANDOM_BOUNCE
        return self

    def plot_search_path(self, pts):
        import matplotlib.pyplot as plt
        import numpy as np
        plt.plot(np.array(pts)[:,0], np.array(pts)[:,1])
        plt.show()
        plt.pause(200)

    def set_search_method_random_bounce(self):
        self._search_method = SearchType.RANDOM_BOUNCE
        self._partitioned_polys = [get_square_poly(self._bounds)]
        return self

    def set_search_method_random_search(self):
        self._search_method = SearchType.RANDOM_SEARCH
        return self

    def set_can_communicate_objective(self, can_communicate_objective):
        self._can_communicate_objective = can_communicate_objective
        return self

    def set_object_found_distance(self, found_distance):
        self._found_distance = found_distance
        return self

    def set_speed(self, speed):
        self._speed = speed
        return self

    def set_starting_position(self, position):
        self._random_position = False
        self._position = position
        return self

    def set_starting_position_random(self):
        self._random_position = True
        return self

    def set_starting_position_on_coordinates(self, coordinates):
        self._starting_position_on_coordinates = True
        self._starting_coordinates = coordinates
        return self

    def set_number_of_drones_to_init(self, num_drones):
        if not isinstance(num_drones, int) and not num_drones.is_integer():
            raise TypeError("Num drones must be an int")
        if num_drones < 1:
            raise ValueError("Num drones must be at least one")
        self._num_drones = num_drones
        return self

    def set_charging_params(self, emergency_recharge_level, set_out_for_seen_trash_while_charging,
     return_to_charge_from_patrolling):
        self._emergency_recharge_level = emergency_recharge_level
        self._set_out_for_seen_trash_while_charging = set_out_for_seen_trash_while_charging
        self._return_to_charge_from_patrolling = return_to_charge_from_patrolling
        # TODO make limits for input parameters
        return self

    def set_trash_detection_radius(self, trash_detection_radius):
        if trash_detection_radius <= 0:
            raise ValueError("Trash detection radius must be positive and non-zero")
        self._trash_detection_radius = trash_detection_radius
        return self

    def commit(self):
        all_drones = []
        self._check_if_can_commit()
        if self._wait_to_start:
            self._index_to_wait_time, self._group_index, distributions = self._get_index_to_wait_time()
            if self._partitioned_lawnmower:
                all_coords = []
                all_polys = []
                for num in distributions:
                    coords_for_drones, polys = partitioned_coords(self._bounds, self.
    _trash_detection_radius, self._speed, num)
                    all_coords.extend(coords_for_drones)
                    all_polys.extend(polys)
                self._all_coords_partitioned_lawnmower = all_coords
                self._partitioned_polys = all_polys
            elif self._search_method == SearchType.RANDOM_BOUNCE:
                if self._partitioned_polys is None:
                    all_polys = []
                    for num in distributions:
                        polys, _ = get_partitions(self._bounds, num)
                        all_polys.extend(polys)
                    self._partitioned_polys = all_polys
                else:
                    self._partitioned_polys *= self._num_drones
            elif self._global_pattern:
                total_waypoints = len(self._patrol_coordinates)
                all_assignments = []
                for num in distributions:
                    start_jump = round(total_waypoints/num)
                    curr = 0
                    for _ in range(num):
                        all_assignments.append(curr)
                        curr += start_jump
        for index in range(self._num_drones):
            drone = Drone(self._bounds)
            if self._random_position:
                self._position = random_position_in_bounds(self._bounds)
            if self._starting_position_on_coordinates:
                # TODO make the starting position on the closest charger to the area it will go to
                self._position = random.choice(self._starting_coordinates)
            if self._potential_fields_on is None:
                self._potential_fields_on = False
            if self._global_pattern:
                drone.start_waypoint = all_assignments[index]
            self._direction = [0, 0]
            self._set_drone_parameters(drone, index)
            self._set_initial_drone_state(drone)
            all_drones.append(drone)
        return all_drones

```

```python
    def _get_index_to_wait_time(self):
        '''Calculates wait times before searching for n groups of drones'''
        charge_fly_ratio = self._recharge_time/self._fly_time
        n = ceil((self._recharge_time + self._fly_time)/self._fly_time)
        if self._num_drones == 1:
            return [0]
        elif self._num_drones == 2:
            second_delay = self._fly_time + self._fly_time/(charge_fly_ratio)
            return [0, second_delay]
        else:
            # Make the first groups get the extra drones
            wait_time = (self._fly_time + self._recharge_time)/n
            base_number = floor(self._num_drones/n)
            leftover = self._num_drones%n
            groups = []
            for _ in range(n):
                if leftover > 0:
                    groups.append(base_number+1)
                    leftover -= 1
                else:
                    groups.append(base_number)
            wait_times = []
            group_index = []
            group_iter = 0
            group_distribution = copy.copy(groups)
            for _ in range(self._num_drones):
                if groups[group_iter] < 1:
                    group_iter += 1
                wait_times.append((group_iter)*wait_time)
                group_index.append(group_iter)
                groups[group_iter] -= 1
            return wait_times, group_index, group_distribution

    def _check_if_can_commit(self):
        if self._fly_time is None:
            raise Exception("Fly time must be set")
        if self._recharge_time is None:
            raise Exception("Recharge time must be set")
        if self._speed is None:
            raise Exception("Speed must be set")
        if self._random_position is None and self._starting_position_on_coordinates is None and self.
    _position is None:
            raise Exception("Position must be initialized")
        if self._num_drones is None:
            raise Exception("Number of drones must be set")
        if self._search_method is None:
            raise Exception("Search method must be set")
        if self._emergency_recharge_level is None:
            raise Exception("Emergency recharge level must be set")
        if self._set_out_for_seen_trash_while_charging is None:
            raise Exception("Set out for trash while charging level must be set")
        if self._return_to_charge_from_patrolling is None:
            raise Exception("Return to charge from patrolling level must be set")
        if self._constant_trash_dropoff_delay is None:
            raise Exception("Dropoff delay must be set")
        if self._constant_trash_pickup_delay is None:
            raise Exception("Pickup delay must be set")

    def _set_initial_drone_state(self, drone):
        if drone.wait_to_start is not None:
            drone._set_state(DroneStateType.WAIT_TO_START, None)
        else:
            drone._set_state(DroneStateType.SEARCH_FOR_TRASH, None)

    def _set_drone_parameters(self, drone, index):
        drone.position = self._position
        drone.direction = self._direction
        drone.speed = self._speed
        drone.fly_time = self._fly_time
        drone.recharge_time = self._recharge_time
        drone.found_distance = self._found_distance
        if self._potential_fields_on:
            drone.avoidance_distance = self._avoidance_distance
            drone.repulse_radius = self._repulse_radius
            drone.attract_scale = self._attract_scale
            drone.set_path_planning_method(PathPlanningType.POTENTIAL_FIELDS)
        else:
            drone.set_path_planning_method(PathPlanningType.DIRECT_ROUTE)
        if self._search_method == SearchType.PATROL:
            if self._partitioned_lawnmower:
                drone.patrol_coordinates = self._all_coords_partitioned_lawnmower[index]
                drone.poly_of_area = self._partitioned_polys[index]
            else:
                drone.patrol_coordinates = self._patrol_coordinates
        if self._wait_to_start:
            drone.wait_to_start = self._index_to_wait_time[index]
            drone.group_index = self._group_index[index]
            if self._search_method == SearchType.RANDOM_BOUNCE:
                drone.poly_of_area = self._partitioned_polys[index]
        drone.set_search_method(self._search_method)
        drone.trash_detection_radius = self._trash_detection_radius
        drone.emergency_recharge_level = self._emergency_recharge_level
        drone.set_out_above_this = self._set_out_for_seen_trash_while_charging
        drone.return_to_charge_from_patrolling = self._return_to_charge_from_patrolling
        drone.can_communicate_objective = self._can_communicate_objective
        drone.trash_pickup_delay = self._constant_trash_pickup_delay
        drone.trash_dropoff_delay = self._constant_trash_dropoff_delay
        return drone
```

clipped_voronoi.py

```
import numpy as np
from shapely.geometry import MultiPoint, Point, Polygon
from scipy.spatial import Voronoi

#Taken from https://gist.github.com/pv/8036995
def voronoi_finite_polygons_2d(vor, radius=None):
    """
    Reconstruct infinite voronoi regions in a 2D diagram to finite
    regions.

    Parameters
    ----------
    vor : Voronoi
        Input diagram
    radius : float, optional
        Distance to 'points at infinity'.

    Returns
    -------
    regions : list of tuples
        Indices of vertices in each revised Voronoi regions.
    vertices : list of tuples
        Coordinates for revised Voronoi vertices. Same as coordinates
        of input vertices, with 'points at infinity' appended to the
        end.

    """

    if vor.points.shape[1] != 2:
        raise ValueError("Requires 2D input")

    new_regions = []
    new_vertices = vor.vertices.tolist()

    center = vor.points.mean(axis=0)
    if radius is None:
        radius = vor.points.ptp().max()

    # Construct a map containing all ridges for a given point
    all_ridges = {}
    for (p1, p2), (v1, v2) in zip(vor.ridge_points, vor.ridge_vertices):
        all_ridges.setdefault(p1, []).append((p2, v1, v2))
        all_ridges.setdefault(p2, []).append((p1, v1, v2))

```

```python
    # Reconstruct infinite regions
    for p1, region in enumerate(vor.point_region):
        vertices = vor.regions[region]

        if all(v >= 0 for v in vertices):
            # finite region
            new_regions.append(vertices)
            continue

        # reconstruct a non-finite region
        ridges = all_ridges[p1]
        new_region = [v for v in vertices if v >= 0]

        for p2, v1, v2 in ridges:
            if v2 < 0:
                v1, v2 = v2, v1
            if v1 >= 0:
                # finite ridge: already in the region
                continue

            # Compute the missing endpoint of an infinite ridge

            t = vor.points[p2] - vor.points[p1] # tangent
            t /= np.linalg.norm(t)
            n = np.array([-t[1], t[0]])  # normal

            midpoint = vor.points[[p1, p2]].mean(axis=0)
            direction = np.sign(np.dot(midpoint - center, n)) * n
            far_point = vor.vertices[v2] + direction * radius

            new_region.append(len(new_vertices))
            new_vertices.append(far_point.tolist())

        # sort region counterclockwise
        vs = np.asarray([new_vertices[v] for v in new_region])
        c = vs.mean(axis=0)
        angles = np.arctan2(vs[:,1] - c[1], vs[:,0] - c[0])
        new_region = np.array(new_region)[np.argsort(angles)]

        # finish
        new_regions.append(new_region.tolist())

    return new_regions, np.asarray(new_vertices)

# Based on https://stackoverflow.com/questions/34968838/python-finite-boundary-voronoi-cells
def generate_clipped_voronoi_diagram_in_square(voronoi_points, min_bounds, max_bounds):
    """ A function that will create a voronoi diagram and clip it in a square

    Arguments:
        min_bounds{float} - lower x,y coordinates for a square
        max_bounds{float} - upper x,y coordinates for a square

    Returns:
        new_polys{list of Shapely polygons} - polygons corresponding to each clipped Voronoi region
        new_vertices{list of numpy arrays with shape 2,N} - vertices of each polygon with the same index
    """
    points_for_convex_hull = np.asarray(
        [[min_bounds, min_bounds],
        [min_bounds, max_bounds],
        [max_bounds,max_bounds],
        [max_bounds,min_bounds]])
    return generate_voronoi_diagram_clipped_in_polygon(voronoi_points, points_for_convex_hull)

def generate_voronoi_diagram_clipped_in_polygon(voronoi_points, points_for_convex_hull):
    """ A function that will create a voronoi diagram and clip it in a convex polygon
    Arguments:
        points{numpy array with shape (2,N)} - N 2D points to construct Voronoi diagram
        points_for_convex_hull{numpy array with shape (2,N)} - vertices of a convex polygon that the function will use to clip the Voronoi region
    """
    vor = Voronoi(voronoi_points)
    # Use a large radius because we are clipping it after
    regions, vertices = voronoi_finite_polygons_2d(vor, radius=100000)

    pts = MultiPoint([Point(i) for i in points_for_convex_hull])
    mask = pts.convex_hull
    new_vertices = []
    new_polys = []
    for region in regions:
        polygon = vertices[region]
        shape = list(polygon.shape)
        shape[0] += 1
        p = Polygon(np.append(polygon, polygon[0]).reshape(*shape)).intersection(mask)
        poly = np.array(list(zip(p.boundary.coords.xy[0][:-1], p.boundary.coords.xy[1][:-1])))
        new_vertices.append(poly)
        new_polys.append(p)
    return new_polys, new_vertices
```

coverage_patterns.py

```python
import copy
from math import sqrt, floor, ceil
import numpy as np
import matplotlib.pyplot as plt
from shapely.geometry import Point, Polygon

from parkcleanup.parkcleanup.tools.coverage_path_generator.clipped_voronoi import
    generate_clipped_voronoi_diagram_in_square
from parkcleanup.parkcleanup.tools.geometry_utils import *
from collector_placement_algorithms.placement_data_utils import load_avgmin_config


def generate_patrol_pattern_for_convex_polygon(polygon, vertices, search_radius):
    '''
    Input is a polygon from shapely (shapely polygon vertices must be in clockwise order)
    and a list of the vertices associated with the polygon,
    and the search radius of the drone.
    '''
    polygon_midpoint = [polygon.centroid.x, polygon.centroid.y]
    distances_to_midpoint = [point_distance(polygon_midpoint, vert) for vert in vertices]
    if max(distances_to_midpoint) < search_radius*2:
        return generate_single_spiral(polygon, polygon_midpoint, vertices, distances_to_midpoint,
    search_radius)
    else:
        return generate_lawnmower_for_convex_polygon(polygon, vertices, search_radius)

def generate_single_spiral(polygon, poly_midpoint, vertices, distances_to_midpoint, search_radius):
    waypoints = []
    for vertice, distance_to_midpoint in zip(vertices, distances_to_midpoint):
        direction_towards_midpoint = vertice - poly_midpoint
        direction_towards_midpoint /= np.linalg.norm(direction_towards_midpoint)
        if distance_to_midpoint < search_radius:
            waypoints.append(poly_midpoint)
        else:
            waypoint = vertice - direction_towards_midpoint*search_radius
            waypoints.append(waypoint)
    return np.asarray(waypoints)

def generate_lawnmower_for_convex_polygon(polygon, vertices, search_radius):
    '''
    Input is a polygon from shapely (make sure the shapely polygon vertices are in clockwise order)
    and a list of the vertices associated with the polygon,
    and the search radius of the drone. Also the axis from matplotlib to plot on.
    '''
    # Make separation radius from the walls smaller than search radius so that drones will see the
    corners and
    # edges of the polygon in between lanes
    offset_from_poly_edges = sqrt(search_radius**2/2)
    # Make the in between the lanes search_radius*2 so there will be less overlap in searching.
    offset_between_lanes = offset_from_poly_edges*2
    polygon_edges = generate_polygon_edges_as_lines(polygon)
    edge_lengths = [line_length(line) for line in polygon_edges]
    # Start the pattern at the longest edge of the polygon and create new lanes in a tangent direction to
     the edge
    longest_edge = np.array(polygon_edges[np.argmax(edge_lengths)])
    t = get_tangent_direction(longest_edge)
    n = get_normal_direction(t)
    midpoint = longest_edge.mean(axis=0)

    # Find vertex that has longest normal distance from the longest polygon edge to determine
    # how many lawnmower lanes to have.
    distances_from_each_vertex_to_longest_edge = [point_to_line_dist(vert, longest_edge) for vert in vertices]
    longest_distance = max(distances_from_each_vertex_to_longest_edge)
    # Now that we know the longest distance, we want to determine how many lanes to make
    # Consider that we want the first and the last lanes to be search_radius distance away from the edges,
    # and the ones in the middle to be at maximum 2*search_radius

    offset_from_poly_edges = offset_from_poly_edges*0.6
    num_line_segments = 1 + round((longest_distance - offset_from_poly_edges*2)/offset_between_lanes)
    num_line_segments = int(num_line_segments)
    start_distance = offset_from_poly_edges
    # Make the distances between lanes equivalent to x, y, y, ... y, x, with x being == offset_from_poly_edges and y <= offset_from_poly_edges*2
    offset_between_lanes = (longest_distance - 2*start_distance)/(num_line_segments-1)

    # Now create all the lines that intersect the polygon
    all_line_information = []
    next_midpoint = midpoint + n*start_distance
    for index in range(num_line_segments):
         # Make huge line with guaranteed intersections in the polygon
        next_line = np.array([next_midpoint+10000*t, next_midpoint-10000*t])
        # Find where this huge line intersects the polygon and also return the edges of the polygon that were intersected
        line_to_add, edge_intersections = get_edge_intersections(next_line, polygon_edges, polygon)
        midpoint = line_to_add.mean(axis=0)
        direction_towards_midpoint = line_to_add[0] - midpoint
        direction_towards_midpoint /= np.linalg.norm(direction_towards_midpoint)
        # Check if the endpoints line up, if not flip it
        if index == 0:
            first_direction = direction_towards_midpoint
        if index != 0:
            dot_product = np.dot(all_line_information[index-1][3], direction_towards_midpoint)
            if dot_product < 0:
                line_to_add = np.flip(line_to_add, axis=0)
                edge_intersections = np.flip(edge_intersections, axis=0).tolist()
                direction_towards_midpoint *= -1
        all_line_information.append((line_to_add, edge_intersections, midpoint, direction_towards_midpoint, [offset_from_poly_edges]))
        next_midpoint = midpoint + n*offset_between_lanes

    # Now move the endpoints of the lines back from the edges
    for (line_to_add, edge_intersections, midpoint, direction_towards_midpoint, line_offset) in
     all_line_information:
        curr_line_length = line_length(line_to_add)
        scale = 1
        if curr_line_length < 2*offset_from_poly_edges:
            while True:
                scale *= 1.00001
                if curr_line_length > 2*offset_from_poly_edges/scale:
                    break
        line_to_add[0] -= offset_from_poly_edges*first_direction/scale
        line_to_add[1] += offset_from_poly_edges*first_direction/scale

    # Now connect everything and plot it
    all_points = []
    all_edge_intersections = []
    for index, (line_to_add, edge_intersections, midpoint, direction_towards_midpoint, line_offset) in
     enumerate(all_line_information):
        even_index_set = index%2==0
        if index != 0 and index != len(all_line_information)-1:
            if not even_index_set:
                # The odd set 1 point is connected with the previous point
                # The direction to offset the newpoint will be the negative of the direction
                points = _find_extra_points_in_between(line_to_add[1], all_points[-1], edge_intersections
     [1], all_edge_intersections[-1], polygon_edges, -first_direction, line_offset[0])
            else:
                # The even set 0 point is connected with the previous point
                points = _find_extra_points_in_between(line_to_add[0], all_points[-1], edge_intersections
     [0], all_edge_intersections[-1], polygon_edges, first_direction, line_offset[0])
            all_points.extend(points)
        if even_index_set:
            all_points.append(line_to_add[0])
            all_points.append(line_to_add[1])
            all_edge_intersections.append(edge_intersections[0])
            all_edge_intersections.append(edge_intersections[1])
        else:
            all_points.append(line_to_add[1])
            all_points.append(line_to_add[0])
            all_edge_intersections.append(edge_intersections[1])
            all_edge_intersections.append(edge_intersections[0])
        # The even set 1 point is connected with the previous 0 point
    all_points = np.asarray(all_points)
    return all_points


def _find_extra_points_in_between(next_point, prev_point, next_edge, prev_edge, all_poly_edges, direction
     , distance):
    '''
    When the lanes are connected, we want the lane connections to follow the contour of the polygon.
    If the lane crossing has a vertice of the polygon outside them, we add a point to help follow
    this vertice.
    '''
    if np.all(np.isclose(next_edge, prev_edge)):
        return []
    else:
        # In the first two options the edges are touching
        if np.all(np.isclose(next_edge[0], prev_edge[1])):
            return [next_edge[0]-direction*distance]
        elif np.all(np.isclose(next_edge[1], prev_edge[0])):
            return [next_edge[1]-direction*distance]
        else:
            points_to_add = []
            next_edge_index = np.argwhere((np.array(all_poly_edges) == next_edge).all(axis=1).all(axis=1)
    ).item(0)
            prev_edge_index = np.argwhere((np.array(all_poly_edges) == prev_edge).all(axis=1).all(axis=1)
    ).item(0)
            # Find the edges in between the ones in question to be connected
            end_of_next_to_begin_of_prev = point_distance(next_edge[1], prev_edge[0])
            begin_of_next_to_end_of_prev = point_distance(next_edge[0], prev_edge[1])
            curr = prev_edge_index
            end = next_edge_index
            if end_of_next_to_begin_of_prev < begin_of_next_to_end_of_prev:
                index_dir = -1
                edge_to_add = 0
            else:
                index_dir = 1
                edge_to_add = 1
            points_to_add.append(prev_edge[edge_to_add]-direction*distance)
            while True:
                curr += index_dir
                if curr >= len(all_poly_edges):
                    curr = 0
                if curr < 0:
                    curr = len(all_poly_edges)-1
                if curr == end:
                    break
                point_to_add = all_poly_edges[curr][edge_to_add]
                point_to_add = point_to_add-direction*distance
                points_to_add.append(point_to_add)
            #points_to_add.reverse()
            return points_to_add

def discretize_paths(discretization, coords, plot=False):
    prev_point = coords[0]
    all_points_to_add = []
    for index, coord in enumerate(coords):
        all_points_to_add.append(prev_point)
        if index == 0:
            prev_point = coord
            continue
        distance = point_distance(coord, prev_point)
        t = get_tangent_direction([coord, prev_point])
        scale = 1
        while True:
            if discretization*scale > distance-3:
                break
            next_point = prev_point + t*discretization*scale
            all_points_to_add.append(next_point)
            scale += 1
        prev_point = coord
    all_points_to_add.append(coords[-1])
    all_points_to_add = np.asarray(all_points_to_add)
    if plot:
        plt.scatter(all_points_to_add[:,0], all_points_to_add[:,1])
    return all_points_to_add

def discretize_paths_with_tdr_and_speed(tdr, speed, coords):
    if tdr/2 < speed*2:
        discretization = speed*2
    else:
        discretization = tdr/2
    return discretize_paths(discretization, coords)

def get_square_poly(bounds):
    vert = [[0,0],[0,bounds],[bounds,bounds],[bounds,0]]
    return Polygon(vert)

def global_lawnmower_coords(bounds, trash_detection_radius, speed):
    vert = [[0,0],[0,bounds],[bounds,bounds],[bounds,0]]
    poly = Polygon(vert)
    coords_2 = generate_patrol_pattern_for_convex_polygon(poly, vert, trash_detection_radius)
    return discretize_paths_with_tdr_and_speed(trash_detection_radius, speed, coords_2).tolist()


def get_partitions(bounds, n):
    if n == 1:
        vert = [[0,0],[0,bounds],[bounds,bounds],[bounds,0]]
        poly = Polygon(vert)
        return [poly], [vert]
    elif n == 2:
        vert1 = [[0,0],[0,bounds],[bounds,bounds],[0,0]]
        poly1 = Polygon(vert1)
        vert2 = [[0,0],[bounds,bounds],[bounds,0],[0,0]]
        poly2 = Polygon(vert2)
        return [poly1, poly2], [vert1, vert2]
    points = load_avgmin_config(n, bounds)
    polys, vertices = generate_clipped_voronoi_diagram_in_square(points, 0, bounds)
    return polys, vertices

def partitioned_coords(bounds, trash_detection_radius, speed, n):
    if n < 3:
        polys, vertices = get_partitions(bounds,n)
    else:
        points = load_avgmin_config(n, bounds)
        polys, vertices = generate_clipped_voronoi_diagram_in_square(points, 0, bounds)
    all_final_coords = []
    for poly, vert in zip(polys, vertices):
        coords = generate_patrol_pattern_for_convex_polygon(poly, vert, trash_detection_radius)
        final_coords = discretize_paths_with_tdr_and_speed(trash_detection_radius, speed, coords).tolist
    ()
        all_final_coords.append(final_coords)
    return all_final_coords, polys

#return generate_multiple_spiral(polygon, polygon_midpoint, vertices, distances_to_midpoint,
    search_radius)
def generate_multiple_spiral(polygon, poly_midpoint, vertices, distances_to_midpoint, search_radius):
    # Not finished, but included for future work
    max_distance_to_midpoint = max(distances_to_midpoint)
    number_spirals = 1+int(round((max_distance_to_midpoint-search_radius)/(search_radius*2)))
    directions_towards_midpoint = []
    vertice_jump_distances = []
    for vertice, distance_to_midpoint in zip(vertices, distances_to_midpoint):
        direction_towards_midpoint = np.asarray(vertice) - np.asarray(poly_midpoint)
        direction_towards_midpoint /= np.linalg.norm(direction_towards_midpoint)
        directions_towards_midpoint.append(direction_towards_midpoint)
        vertice_jump_distances.append(distance_to_midpoint/number_spirals)
    waypoints = []
    for i in range(number_spirals):
        for vert, direction, jump_distance in zip(vertices, directions_towards_midpoint,
    vertice_jump_distances):
            waypoints.append(vert - direction*(search_radius*2*(i)+search_radius))

    get_path_length(waypoints)
    return waypoints

def get_path_length(waypoints):
    start = True
    total = 0
    for point in waypoints:
        if start:
            prev_waypoint = point
            start = False
        else:
            total += point_distance(prev_waypoint, point)
            prev_waypoint = point
    print(total)

if __name__ == "__main__":
    sr=10
    disc = sr
    park_len = 100
    # #n=1
    fig, ax = plt.subplots()
    vert = [[0,0],[0,100],[100,100],[100,0]]
    ax.set_xlim(0,park_len*1.1)
    ax.set_ylim(0,park_len*1.1)
    poly = Polygon(vert)
    coords_2 = generate_patrol_pattern_for_convex_polygon(poly, vert, disc)
    ax.plot(coords_2[:,0], coords_2[:,1])
    # final_coords = discretize_paths(20, coords_2)
    plt.show()

    # #n=2
    fig, ax = plt.subplots()
    square =np.array([[0,0],[0,100],[100,100],[100,0],[0,0]])
    ax.plot(square[:,0], square[:,1])
    ax.set_xlim(0,100)
    ax.set_ylim(0,100)
    vert1 = [[0,0],[0,100],[100,100],[0,0]]
    poly1 = Polygon(vert1)
    coords = generate_patrol_pattern_for_convex_polygon(poly1, vert1, disc)
    final_coords = discretize_paths(disc/2, coords, plot=False)
    for drone_position in final_coords:
        circle = plt.Circle((drone_position[0],drone_position[1]), sr, color='b')
        ax.add_artist(circle)

    vert2 = [[0,0],[100,100],[100,0],[0,0]]
    poly2 = Polygon(vert2)
    coords = generate_patrol_pattern_for_convex_polygon(poly2, vert2, disc)
    final_coords = discretize_paths(disc/2, coords, plot=False)
    for drone_position in final_coords:
        circle = plt.Circle((drone_position[0],drone_position[1]), sr, color='b')
        ax.add_artist(circle)
    ax.set_xlim(0,100)
    ax.set_ylim(0,100)
    plt.show()

    for i in range(3, 20):
        fig, ax = plt.subplots()
        points = load_avgmin_config(i, bounds)
        polys, vertices = generate_clipped_voronoi_diagram_in_square(points, 0, 100)
        for poly, vert in zip(polys, vertices):
            coords = generate_patrol_pattern_for_convex_polygon(poly, vert, disc)
            final_coords = np.vstack((coords, coords[0]))
            final_coords = discretize_paths(disc/2, final_coords, plot=False)
            final_coords = final_coords[:-1]
            for drone_position in final_coords:
                circle = plt.Circle((drone_position[0],drone_position[1]), sr, color='b', alpha=0.5)
                ax.add_artist(circle)
        plt.show()
```

geometry_utils.py

```
import copy
import math
from math import sqrt
import numpy as np
from collections import namedtuple

import numpy as np
from shapely.geometry import Point

def generate_polygon_edges_as_lines(p):
    vertices = p.exterior.coords.xy
    all_lines = []
    for i in range(len(vertices[0])):
        point = [vertices[0][i], vertices[1][i]]
        if i==0:
            prev = copy.deepcopy(point)
            continue
        else:
            line = [prev, point]
            all_lines.append(line)
            prev = copy.deepcopy(point)
    return all_lines


def closest_edge(edges, point):
    distances = [point_to_line_dist(point, edge, normal_or_closest_endpoint=True) for edge in edges]
    return distances.index(min(distances))


def line_length(line):
    point1 = line[0]
    point2 = line[1]
    return sqrt((point2[0]- point1[0])**2 + (point2[1]- point1[1])**2)


def point_distance(point1, point2):
```

```python
    return sqrt((point2[0]- point1[0])**2 + (point2[1]- point1[1])**2)


def get_tangent_direction(line):
    t = line[0] - line[1] # x and y components of slope
    t /= np.linalg.norm(t)
    return t


def get_normal_direction(tangent):
    return np.array([-tangent[1], tangent[0]])


def get_edge_intersections(next_line, edge_lines, p):
    intersection_points = []
    edges_that_intersected = []
    for edge in edge_lines:
        result = find_line_intersection(next_line, edge)
        if result[2] == 0:
            continue
        point = Point(result[0], result[1])
        to_store = [result[0], result[1]]
        if np.isclose(p.distance(point), 0):
            intersection_points.append(to_store)
            edges_that_intersected.append(edge)
    line_to_add = np.array(intersection_points)
    return line_to_add, edges_that_intersected


#https://stackoverflow.com/questions/27161533/find-the-shortest-distance-between-a-point-and-line-
    segments-not-line
def point_to_line_dist(point, line, normal_or_closest_endpoint=False):
    """Calculate the distance between a point and a line segment.
    If normal_or_closest endpoint is false, it returns the perpendicular distance from the line extended
    infinitely to the point.
    If it is true, this wlil return either perpendicular distance or if the point cannot trace a
    perpendicular line back to the point,
    the closest to one of the endpoints.
    """
    Point = namedtuple('Point', ['x', 'y'])
    a = Point(line[0][0], line[0][1])
    b = Point(line[1][0], line[1][1])
    other_point = Point(point[0], point[1])
    dx = b.x - a.x
    dy = b.y - a.y
    dr2 = float(dx ** 2 + dy ** 2)

    lerp = ((other_point.x - a.x) * dx + (other_point.y - a.y) * dy) / dr2
    if normal_or_closest_endpoint:
        if lerp < 0:
            lerp = 0
        elif lerp > 1:
            lerp = 1

    x = lerp * dx + a.x
    y = lerp * dy + a.y

    _dx = x - other_point.x
    _dy = y - other_point.y
    square_dist = _dx ** 2 + _dy ** 2
    return np.sqrt(square_dist)


# From https://www.cs.hmc.edu/ACM/lectures/intersections.html
def find_line_intersection(line1, line2):
    """ this returns the intersection of Line(pt1,pt2) and Line(ptA,ptB)

        returns a tuple: (xi, yi, valid, r, s), where
        (xi, yi) is the intersection
        r is the scalar multiple such that (xi,yi) = pt1 + r*(pt2-pt1)
        s is the scalar multiple such that (xi,yi) = pt1 + s*(ptB-ptA)
            valid == 0 if there are 0 or inf. intersections (invalid)
            valid == 1 if it has a unique intersection ON the segment    """
    pt1, pt2, ptA, ptB = line1[0], line1[1], line2[0], line2[1]
    DET_TOLERANCE = 0.00000001

    # the first line is pt1 + r*(pt2-pt1)
    # in component form:
    x1, y1 = pt1
    x2, y2 = pt2
    dx1 = x2 - x1
    dy1 = y2 - y1

    # the second line is ptA + s*(ptB-ptA)
    x, y = ptA
    xB, yB = ptB
    dx = xB - x
    dy = yB - y

    # we need to find the (typically unique) values of r and s
    # that will satisfy
    #
    # (x1, y1) + r(dx1, dy1) = (x, y) + s(dx, dy)
    #
    # which is the same as
    #
    #    [ dx1  -dx ][ r ] = [ x-x1 ]
    #    [ dy1  -dy ][ s ] = [ y-y1 ]
    #
    # whose solution is
    #
    #    [ r ] = _1_  [  -dy   dx ] [ x-x1 ]
    #    [ s ] = DET  [ -dy1  dx1 ] [ y-y1 ]
    #
    # where DET = (-dx1 * dy + dy1 * dx)
    #
    # if DET is too small, they're parallel
    #
    DET = (-dx1 * dy + dy1 * dx)

    if math.fabs(DET) < DET_TOLERANCE: return (0,0,0,0,0)

    # now, the determinant should be OK
    DETinv = 1.0/DET

    # find the scalar amount along the "self" segment
    r = DETinv * (-dy  * (x-x1) +  dx * (y-y1))

    # find the scalar amount along the input line
    s = DETinv * (-dy1 * (x-x1) + dx1 * (y-y1))

    # return the average of the two descriptions
    xi = (x1 + r*dx1 + x + s*dx)/2.0
    yi = (y1 + r*dy1 + y + s*dy)/2.0
    return ( xi, yi, 1, r, s )


def testIntersection( pt1, pt2, ptA, ptB ):
    """ prints out a test for checking by hand... """
    print("Line segment #1 runs from", pt1, "to", pt2)
    print("Line segment #2 runs from", ptA, "to", ptB)

    result = find_line_intersection( pt1, pt2, ptA, ptB )
    print("    Intersection result =", result)


if __name__ == "__main__":

  pt1 = (10,10)
  pt2 = (20,20)

  pt3 = (10,20)
  pt4 = (20,10)

  pt5 = (40,20)

  testIntersection( pt1, pt2, pt3, pt4 )
  testIntersection( pt1, pt3, pt2, pt4 )
  testIntersection( pt1, pt2, pt4, pt5 )
```

helper.py

```
import math
from random import uniform

from numpy import std

def sign(x):
    return math.copysign(1, x)

def random_position_in_bounds(bounds):
    return [uniform(0, bounds), uniform(0, bounds)]

def distance(p1, p2):
    x1 = p1[0]
    x2 = p2[0]
    y1 = p1[1]
    y2 = p2[1]
    return math.sqrt((x2-x1)**2+(y2-y1)**2)

def mean(data):
    mean = 0
    for value in data:
        mean += value
    return mean/len(data)

def std_dev(data):
    if len(data) == 1:
        return 0
    else:
        return std(data)
```

movable.py

```
from math import sin
from math import cos
from math import sqrt
from math import exp
from random import random as rand

from parkcleanup.parkcleanup.tools.helper import sign

class Movable(object):
    def __init__(self, position, direction, speed, repulse_radius=None, attract_scale=None):
        self.position = position
        self.direction = direction
        self.speed = speed
        self.repulse_radius = repulse_radius
        self.attract_scale = attract_scale

    def _rotate_vector(self, angle):
        vector = self.direction
        x2 = cos(angle)*vector[0]-sin(angle)*vector[1]
        y2 = sin(angle)*vector[0]+cos(angle)*vector[1]
        vector[0] = x2
        vector[1] = y2
        self.direction = self.normalize_vector(vector[0], vector[1])

    def _update_direction_from_objective_straight_line(self, objective):
        direction = self.calculate_direction(self.position,objective)
        self.direction = self.normalize_vector(direction[0],direction[1])

    def _potential_fields_update_direction(self, things_we_are_trying_to_avoid, objective=None):
        curr_coords = self.position
        repulse_force_x = 0
        repulse_force_y = 0
        x = curr_coords[0]
        y = curr_coords[1]
        forces = []
        if len(things_we_are_trying_to_avoid) != 0:
            for things in things_we_are_trying_to_avoid:
                xdist = x-things[0]
                ydist = y-things[1]
                repulse_force_x += sign(xdist)*exp(-1/2*(xdist/self.repulse_radius)**2)
                repulse_force_y += sign(ydist)*exp(-1/2*(ydist/self.repulse_radius)**2)
                forces.append([repulse_force_x,repulse_force_y])
        if objective != None:
            attract_force_x = objective[0]-x
            attract_force_y = objective[1]-y
            attract_force = [attract_force_x*self.attract_scale, attract_force_y*self.attract_scale]
            forces.append(attract_force)
        final_force_x = 0
        final_force_y = 0
        for force in forces:
            final_force_x += force[0]
            final_force_y += force[1]
        final_force = self.normalize_vector(final_force_x, final_force_y)
        self.direction = final_force

    def _update_coordinates(self):
        x_coord = (self.position[0] + self.direction[0]*self.speed)
        y_coord = (self.position[1] + self.direction[1]*self.speed)
        self.position = [x_coord,y_coord]

    @staticmethod
    def distance(p1,p2):
        return sqrt((p2[0]-p1[0])**2+(p2[1]-p1[1])**2)

    @staticmethod
    def calculate_direction(p1,p2):
        return [p2[0]-p1[0], p2[1]-p1[1]]

    @staticmethod
    def normalize_vector(x, y):
        magnitude = sqrt(x**2 + y**2)
        if magnitude == 0:
            return [0, 0]
        else:
            return [x/magnitude, y/magnitude]
```

drone_state_type.py

```
from enum import Enum


class DroneStateType(Enum):
    GO_TO_TRASH = "Go to Trash"
    GO_TO_COLLECTOR = "Go to Collector"
    SEARCH_FOR_TRASH = "Search for Trash"
    GO_TO_CHARGER = "Go to Charger"
    RECHARGE = "Recharge"
    DROP_OFF_TRASH = "Drop off Trash"
    PICK_UP_TRASH = "Pick up Trash"
    OUT_OF_ENERGY = "Out of Energy"
    WAIT_TO_START = "Wait to start"
    TAKE_OFF = "Take off"
    LAND_ON_CHARGER = "Land on charger"
```

drone_path_planning_strategies.py

```
import abc

class _PathPlanningStrategy(abc.ABC):
    def __init__(self):
        pass

    def update_direction(self, drone, sim_model):
        pass

class _PotentialFields(_PathPlanningStrategy):
    def __init__(self):
        pass

    def update_direction(self, drone, sim_model):
        people_we_are_trying_to_avoid = []
        if sim_model.there_are_people_in_model():
            all_distances_from_persons = sim_model.drone_to_person[drone.id]
            for index, distance in enumerate(all_distances_from_persons):
                if distance < drone.avoidance_distance:
                    people_we_are_trying_to_avoid.append(sim_model.person_coords[index])
        all_distances_from_drones = sim_model.drone_to_drone[drone.id]
        drones_we_are_trying_to_avoid = []
        for index, distance in enumerate(all_distances_from_drones):
            if distance < drone.avoidance_distance:
                drones_we_are_trying_to_avoid.append(sim_model.drone_coords[index])

        things_we_are_trying_to_avoid = people_we_are_trying_to_avoid + drones_we_are_trying_to_avoid
        drone._potential_fields_update_direction(things_we_are_trying_to_avoid, objective=drone.objective
    .position)

class _DirectRoute(_PathPlanningStrategy):
    def __init__(self):
        pass

    def update_direction(self, drone, sim_model):
        drone._update_direction_from_objective_straight_line(drone.objective.position)
```

drone_search_strategies.py

```
import abc
from random import randint
from random import random
from random import choice

from scipy.spatial import distance_matrix
from shapely.geometry import Point
import numpy as np

from parkcleanup.parkcleanup.model.objectives.location import Location
from parkcleanup.parkcleanup.tools.helper import random_position_in_bounds
from parkcleanup.parkcleanup.tools.coverage_path_generator.coverage_patterns import get_square_poly
from parkcleanup.parkcleanup.tools.geometry_utils import generate_polygon_edges_as_lines, line_length,
    get_tangent_direction, closest_edge

class _SearchStrategy(metaclass=abc.ABCMeta):
    def __init__(self):
        pass

    def update_strategy_on_state_change(self, drone, sim_model):
```

```python
        pass

    def search_update_method(self, drone, sim_model):
        pass

class _RandomSearch(_SearchStrategy):
    def __init__(self):
        pass

    def update_strategy_on_state_change(self, drone, sim_model):
        pass

    def search_update_method(self, drone, sim_model):
        x = random()*drone.speed*2-drone.speed+drone.position[0]
        y = random()*drone.speed*2-drone.speed+drone.position[1]
        drone.objective = Location([x, y])

class _PatrolSearch(_SearchStrategy):
    def __init__(self, patrol_coordinates, closest_waypoint_on_resume):
        self.patrol_coordinates = patrol_coordinates
        self.closest_waypoint_on_resume = closest_waypoint_on_resume
        self.wait_for_one = True
        self._patrol_index = None

    def update_strategy_on_state_change(self, drone, sim_model):
        if self.closest_waypoint_on_resume:
            distances_to_locations = distance_matrix([drone.position], self.patrol_coordinates).tolist()
            index_of_min_distance_location = distances_to_locations[0].index(min(distances_to_locations[0]))
            self._patrol_index = index_of_min_distance_location
            drone.objective = Location(self.patrol_coordinates[index_of_min_distance_location])
        else:
            if self._patrol_index == None:
                if drone.start_waypoint is None:
                    distances_to_locations = distance_matrix([drone.position], self.patrol_coordinates).tolist()
                    index_of_min_distance_location = distances_to_locations[0].index(min(distances_to_locations[0]))
                    self._patrol_index = index_of_min_distance_location
                    drone.objective = Location(self.patrol_coordinates[index_of_min_distance_location])
                else:
                    self._patrol_index = drone.start_waypoint
                    drone.objective = Location(self.patrol_coordinates[drone.start_waypoint])
            else:
                drone.objective = Location(self.patrol_coordinates[self._patrol_index])

    def search_update_method(self, drone, sim_model):
        if drone._reached_objective():
            # TODO figure out the interaction effects with reaching a goal and change this function
            # drone.position = drone.objective.position
            # self._set_next_location_objective(drone)
            if self.wait_for_one:
                self.wait_for_one = False
            else:
                drone.position = drone.objective.position
                self._set_next_location_objective(drone)
                self.wait_for_one = True

    def _set_next_location_objective(self, drone):
        index = self._patrol_index
        index += 1
        if index == len(self.patrol_coordinates):
            index = 0
        self._patrol_index = index
        drone.objective = Location(self.patrol_coordinates[index])


class _RandomBounceSearch(_SearchStrategy):
    def __init__(self, poly, bounds):
        self._side = None
        if poly is None:
            poly = get_square_poly(bounds)
        self._poly = poly
        self._edges = generate_polygon_edges_as_lines(poly)
        self._num_sides = len(self._edges)
        self._in_poly = None

    def update_strategy_on_state_change(self, drone, sim_model):
        if not self._poly.contains(Point(drone.position[0], drone.position[1])):
            self._in_poly = False
            centroid = self._poly.centroid.coords.xy
            centroid = [centroid[0][0], centroid[1][0]]
            drone.objective = Location(centroid)
        else:
            random_side = randint(0, self._num_sides-1)
            self._side = random_side
            self._in_poly = True
            drone.objective = Location(self._random_edge_in_poly(sim_model.park.bounds, random_side))

    def search_update_method(self, drone, sim_model):
        if not self._in_poly:
            if self._poly.contains(Point(drone.position[0], drone.position[1])):
                self._in_poly = True
                drone.objective = Location(drone.position)
                self._side = closest_edge(self._edges, drone.position)
            else:
                return
        if drone._reached_objective():
            random_sides = list(range(self._num_sides))
            random_sides.remove(self._side)
            random_side = choice(random_sides)
            self._side = random_side
            drone.objective = Location(self._random_edge_in_poly(self._poly, random_side))

    def _random_edge_in_poly(self, poly, random_side):
        line = np.array(self._edges[random_side])
        length = line_length(line)
        tangent = get_tangent_direction(np.flip(line, axis=0))
        rand_in_bounds = random()*length
        return (line[0] + tangent*rand_in_bounds).tolist()
```

A.1.2 Visualization

plotter.py

```
from abc import ABC, abstractmethod

import matplotlib
import matplotlib.path as mplPath
from matplotlib import pyplot as plt

class Plotter(ABC):
    def __init__(self):
        self._title_on = False
        self._title = None

        self._person_scatter = None
        self._drone_scatter = None
        self._trash_scatter = None
        self._collector_scatter = None
        self._charger_scatter = None
        self._end_time_step = None
        self._start_time_step = None
        self._trash_per_time_step_on = False
        self._extra_plots = 0
        self._show_trash_detection_radius_circle = False
        self._drone_color_change_battery_level_on = False
        self._show_drone_search_pattern = False

        self._show_inputs = False
        self._input_dict = None

    def show_inputs(self, input_dict):
```

```python
        if input_dict is not None:
            self._show_inputs = True
            self._input_dict = input_dict

    def show_drone_search_patterns(self):
        self._show_drone_search_pattern = True
        return self

    def set_drone_color_change_for_battery_level(self):
        self._drone_color_change_battery_level_on = True
        return self

    def show_trash_detection_radius_circle(self):
        self._show_trash_detection_radius_circle = True
        return self

    def show_when_trash_is_identified_with_color(self):
        pass

    def set_title(self, title):
        self._title_on = True
        if not isinstance(title, str):
            raise TypeError("Title must be string")
        self._title = title
        return self

    def step_is_at_least_min_time_step(self, curr_time_step):
        if self._start_time_step is None:
            return True
        return curr_time_step > self._start_time_step

    def set_start_timestep_for_plotting(self, start_time_step):
        if not isinstance(start_time_step, int):
            raise TypeError("Start timestep must be int")
        if start_time_step < 0:
            raise ValueError("Start timestep must be positive")
        self._start_time_step = start_time_step

    def set_end_timestep_for_plotting(self, end_time_step):
        if not isinstance(end_time_step, int):
            raise TypeError("Max timestep must be int")
        if end_time_step < 1:
            raise ValueError("Max timestep must be positive and nonzero")
        self._end_time_step = end_time_step

    def show_outputs(self):
        self._show_outputs = True
        return self

    @abstractmethod
    def init_plot(self, park_sim, has_run):
        pass

    @abstractmethod
    def update_plot(self, data_logger, time_step):
        pass

    @abstractmethod
    def close_plot(self):
        pass

    def interactive_plot_data(self, park_sim, show=True):
        if not park_sim.has_run():
            raise Exception("Sim cannot be plotted because it has not been run yet")
        self.init_plot(park_sim, has_run=True, show=show)

    def plot_data(self, park_sim):
        if not park_sim.has_run():
            raise Exception("Sim cannot be plotted because it has not been run yet")
        self.init_plot(park_sim, has_run=False)
        for i in range(park_sim.num_time_steps):
            if not self.step_is_at_least_min_time_step(i):
                continue
            self.update_plot(park_sim.sim_model, i)
            if self._end_time_step_reached(i):
                break
        self.close_plot()

    def _end_time_step_reached(self, time_step):
        if self._end_time_step is not None:
            if self._end_time_step == time_step:
                return True
        return False
```

matplotlib_plotter.py

```
import decimal
from math import floor, ceil
import time

import numpy as np
import matplotlib as mpl
import matplotlib.path as mplPath
from matplotlib import pyplot as plt
from mpl_toolkits.axes_grid1 import Divider, Size, make_axes_locatable
from matplotlib.widgets import Slider, Button, RadioButtons, TextBox

from parkcleanup.parkcleanup.visualization.plotter import Plotter
from parkcleanup.parkcleanup.model.agents.drone_state_type import *
from experiment_runner.experiment_runner.string_constants import *

LAST_VISITED_HEATMAP_RADIO_TEXT = "UAV HM"
TRASH_LEFT_OUT_HEATMAP = "Trash HM"
OFF = "Off"

class MatplotlibPlotter(Plotter):
    def __init__(self):
        super().__init__()
        self._speed = 0.00001
        self.all_circles = []

    def init_plot(self, park_sim, has_run, show=True):
        sim_model = park_sim.sim_model
        # Set plotting settings
        mpl.rc('font', **{'sans-serif' : 'Arial',
                          'family' : 'sans-serif'})
        self.curr_index = 0
        self.sim_model = sim_model
        data_logger = park_sim.data_logger
        self.hm_at_every_time_step = data_logger.hm_at_every_time_step
        # Initialize heatmap and colorbar stuff
        self._drone_heatmap = None
        self._trash_heatmap = None
        self._drone_colorbar = None
        self._trash_colorbar = None
        self._heat_map_value_selected = OFF
        # Make the primary update method do nothing since OFF is selected
        self._heat_map_data_update_method = lambda a, b: None
        self._cax = None
        self._vmax = 1000
        # Initialize variables that will be referenced
        self._trash_detection_radius = park_sim.sim_model.all_drones[0].trash_detection_radius

        # Initialize plots
        fig, axes = plt.subplots(1,2,figsize=(15, 5),dpi=100)
        # Make fig a little bit smaller to fit more widgets later
        fig.subplots_adjust(left=0.1,right=0.85,bottom=0.1,top=0.9)
        self._main_ax = axes[0]
        self._data_axis = axes[1]
        self._plot_trash_per_time_step_plot(self._data_axis, sim_model, data_logger)
        self._fig = fig

        # Plot park with some visual cushion on the outside
        side_length = sim_model.park.bounds
        self._side_length = side_length
        self._main_ax.set_xlim(-side_length*0.1, side_length*1.1)
        self._main_ax.set_ylim(-side_length*0.1, side_length*1.1)
        self._plot_outside_bounds(side_length, self._main_ax)

        if self._show_inputs:
            self._plot_the_inputs(self._main_ax)
        if self._show_outputs:
            self._plot_the_outputs(self._main_ax, sim_model, data_logger)
        if sim_model.park.nodes_on:
            self._plot_park_paths(sim_model, self._main_ax)

        # Initialize plots that will be updated
        x, y = [],[]
        self._person_scatter = self._main_ax.scatter(x, y)
        self._drone_scatter = self._main_ax.scatter(x, y, marker='$\xa4$', cmap="Greys", vmin=0, vmax=1,
    s=100, alpha=0.9)
        self._trash_scatter = self._main_ax.scatter(x, y, marker="X", color="r", s=100)
        self._collector_scatter = self._main_ax.scatter(x, y, marker=r'$\sqcup$', color="saddlebrown", s
    =100)
        self._charger_scatter = self._main_ax.scatter(x, y, marker="P", color="m", s=100)
        # Initialize the data for the scatter plot that changes the color of the trash that has been left
     out the longest
        max_trash_indices = data_logger.get_max_trash_indices()

        self._all_max_trash_indices = max_trash_indices
        self._single_longest_trash_scatter = self._main_ax.scatter(x, y, marker="X", color="g")

        if self._title_on:
            self._main_ax.set_title(self._title)
        self._minute_time_text = self._main_ax.text(1.01, 0.97, '', transform=self._main_ax.transAxes)
        self._hour_time_text = self._main_ax.text(1.01, 0.94, '', transform=self._main_ax.transAxes)
        self._main_ax.text(2.45, -0.1, 'seconds', transform=self._main_ax.transAxes)
        # Its hard to know which group of drones is out when
        # TODO make the group number be accurate
        # self._group_number_text = self._main_ax.text(1.1, 0.7, 'Group: 0', transform=self._main_ax.
    transAxes)

        # Make the legend appear outside of the plot
        box = self._main_ax.get_position()
        self._main_ax.set_position([box.x0+0.04, box.y0, box.width * 0.8, box.height])
        self._main_ax.legend((self._drone_scatter, self._trash_scatter, self._collector_scatter, self.
    _charger_scatter),
        ("UAVs","Trash","Collectors","Chargers"), bbox_to_anchor=(1.01,0.4),loc='center left')
        if self._drone_color_change_battery_level_on:
        # Manually set the drone legend color to gray because setting the marker color to gray
        # in the drone scatter plot initialization prevents the battery level color change effect from
    happening
            self._main_ax.get_legend().legendHandles[0].set_color('gray')

        # Trash heatmaps
        heat_maps = data_logger.get_average_time_trash_in_cell_hms()
        num_trash_heat_map = data_logger.get_num_trash_collected_heat_map()
        all_drone_heat_maps = data_logger.get_all_last_search_heat_map()
        avg_heat_map = data_logger.get_average_heat_map()
        num_times_visited = data_logger.get_num_times_visited_hm()

        if self.hm_at_every_time_step:
            self._all_trash_heat_maps = heat_maps
            self._max_trash_heat_map = int(np.max(heat_maps))

            self._all_drone_heat_maps = all_drone_heat_maps
            self._max_drone_heatmap = int(np.max(all_drone_heat_maps))

        self._num_trash_heat_map = num_trash_heat_map
        self._max_num_trash_heat_map = int(np.max(num_trash_heat_map))

        self._average_drone_heat_map = avg_heat_map
        # The max usually has a crazy amount of decimals, so round it
        self._max_average_drone_heat_map = round(np.max(self._average_drone_heat_map), 2)

        self._number_times_visited = num_times_visited
        self._max_number_times_visited = int(np.max(num_times_visited))

        self._all_max = data_logger.all_max_hm
        self._all_mean = data_logger.all_mean_hm
        self._all_std_dev = data_logger.all_std_dev_hm

        # Interactive update things
        # Create all the buttons and widgets
        # The axes arguments are: x position, y position, x length, y length
        axcolor = 'lightgoldenrodyellow'
        axfreq = plt.axes([0.1, 0.01, 0.65, 0.03], facecolor=axcolor)
        f0=0
        delta_f = 1
        time_step_update_slider = Slider(axfreq, 'Time Step', 0, data_logger.num_time_steps, valinit=f0,
    valstep=delta_f)

        start_button_placeholder = plt.axes([0.005, 0.2, 0.025, 0.04])
        start_button = Button(start_button_placeholder, 'Play', color=axcolor, hovercolor='0.975')

        pause_button_placeholder = plt.axes([0.03, 0.2, 0.033, 0.04])
        pause_button = Button(pause_button_placeholder, 'Pause', color=axcolor, hovercolor='0.975')

        back_button_placeholder = plt.axes([0.068, 0.2, 0.029, 0.04])
        back_button = Button(back_button_placeholder, 'Back', color=axcolor, hovercolor='0.975')

```

```
        next_button_placeholder = plt.axes([0.097, 0.2, 0.029, 0.04])
        next_button = Button(next_button_placeholder, 'Next', color=axcolor, hovercolor='0.975')

        axbox = plt.axes([0.05, 0.1, 0.05, 0.075])
        text_box = TextBox(axbox, 'Jump To:', initial="0")

        axbox_vmax = plt.axes([0.45, 0.55, 0.04, 0.075])
        text_box_vmax = TextBox(axbox_vmax, 'vmax:', initial=str(self._vmax))
        self._text_box_vmax = text_box_vmax
        self._use_default_vmax = True

        axbox = plt.axes([0.03, 0.27, 0.05, 0.075])
        speed_slider = Slider(axbox, 'Speed', 1, 40, valinit=1, valstep=delta_f)

        rax = plt.axes([0.42, 0.1, 0.05, 0.2], facecolor=axcolor)
        drone_pattern_radio = RadioButtons(rax, (0, 1, 2), active=0)

        axbox_for_output_radio = plt.axes([0.83, 0.24, 0.15, 0.2], facecolor=axcolor)
        data_output_radio = RadioButtons(axbox_for_output_radio, (
            TOTAL_TRASH,
            AVG_TRASH_LEFT_OUT,
            LONGEST_CURRENT_TRASH,
            AVG_TIME_TRASH_LEFT_OUT,
            MAX_TIME_SINCE_VISITED,
            AVG_TIME_SINCE_VISITED,
            STD_DEV_TIME_SINCE_VISITED,
            ), active=0)

        axbox_for_heat_map_radio = plt.axes([0.42, 0.65, 0.093, 0.17], facecolor=axcolor)
        heat_map_radio = RadioButtons(axbox_for_heat_map_radio, (
            OFF,
            LAST_VISITED_HEATMAP_RADIO_TEXT,
            TRASH_LEFT_OUT_HEATMAP,
            NUMBER_TIMES_VISITED,
            AVERAGE_VISITED,
            NUM_TOTAL_TRASH
            ), active=0)

        # Create all the update methods for when the widgets are activated (by button press, text enter,
    etc.)
        def update_drone_patterns(label):
            self.patrol_plots = []
            self.partition_plots = []
            for drone in sim_model.all_drones:
                if drone.group_index != label:
                    continue
                if drone.patrol_coordinates is not None:
                    coords_to_plot = np.array(drone.patrol_coordinates)
                    self.patrol_plots.append(self._main_ax.plot(coords_to_plot[:,0], coords_to_plot[:,1], alpha=0.5))
                if drone.poly_of_area is not None:
                    self.partition_plots.append(self._main_ax.plot(*drone.poly_of_area.exterior.xy, c='g', alpha=0.5))

        def update_heat_map_vmax(value):
            self._vmax = value
            self._use_default_vmax = False
            value_selected = heat_map_radio.value_selected
            update_map_background_plot(value_selected)
            self._use_default_vmax = True

        def update_map_background_plot(label):
            if label == LAST_VISITED_HEATMAP_RADIO_TEXT:
                self._plot_last_visited_step_heatmap(self.sim_model, self.curr_index)
            elif label == TRASH_LEFT_OUT_HEATMAP:
                self._plot_weighted_trash_per_time_step_heatmap(self.sim_model, self.curr_index)
            elif label == OFF:
                self._clear_heat_maps()
                self._heat_map_data_update_method = lambda a, b: None
            elif label == NUMBER_TIMES_VISITED:
                self._plot_number_of_times_visited(sim_model)
            elif label == AVERAGE_VISITED:
                self._plot_average_heat_map_value(sim_model)
            elif label == NUM_TOTAL_TRASH:
                self._plot_num_trash_heat_map(sim_model)
            self._fig.canvas.draw_idle()

        def update_output_plot(label):
            if label == TOTAL_TRASH:
                self._plot_trash_per_time_step_plot(self._data_axis, self.sim_model, data_logger)
            elif label == LONGEST_CURRENT_TRASH:
                self._plot_max_time_left_out_in_each_time_step_plot(self._data_axis, self.sim_model, data_logger)
            elif label == AVG_TIME_TRASH_LEFT_OUT:
                self._plot_avg_time_trash_left_out_in_each_time_step_plot(self._data_axis, self.sim_model, data_logger)
            elif label == AVG_TRASH_LEFT_OUT:
                self._plot_avg_trash_left_out_in_each_time_step_plot(self._data_axis, sim_model, data_logger)
            elif label == AVG_TIME_SINCE_VISITED:
                self._plot_avg_since_last_visited_plot(self._data_axis, sim_model, data_logger)
            elif label == MAX_TIME_SINCE_VISITED:
                self._plot_max_since_last_visited_plot(self._data_axis, sim_model, data_logger)
            elif label == STD_DEV_TIME_SINCE_VISITED:
                self._plot_std_dev_since_last_visited_plot(self._data_axis, sim_model, data_logger)
            elif label == ACTIVE_RATIO:
                self._plot_active_ratios_plot(self._data_axis, sim_model, data_logger)
            self._fig.canvas.draw_idle()

        def drone_pattern_radio_update(label):
            for patrol_plot_set in self.patrol_plots:
                for patrol_plot in patrol_plot_set:
                    patrol_plot.remove()
            for partition_plot_set in self.partition_plots:
                for partition_plot in partition_plot_set:
                    partition_plot.remove()
            self._fig.canvas.draw_idle()
            update_drone_patterns(int(label))

        def update_speed_box(val):
            val = int(val)
            self._play_speed = val

        def update_slider(val):
            val = int(val)
            if val < data_logger.num_time_steps:
                self.update_plot(sim_model, val, data_logger)

        def update_text_box(val):
            val = int(val)
            if val < data_logger.num_time_steps:
                time_step_update_slider.set_val(val)
        self.stop = False

        def back_button_update(event):
            val = time_step_update_slider.val
            if val - self._play_speed >= 0:
                time_step_update_slider.set_val(int(val - self._play_speed))
            else:
                time_step_update_slider.set_val(0)

        def next_button_update(event):
            val = time_step_update_slider.val
            if val + self._play_speed < data_logger.num_time_steps:
                time_step_update_slider.set_val(int(val + self._play_speed))
            else:
                time_step_update_slider.set_val(data_logger.num_time_steps-1)

        def pause_button_update(event):
            self.stop = True

        def play_button_update(event):
            self.stop = False
            while not self.stop:
                val = time_step_update_slider.val
                if val+self._play_speed > data_logger.num_time_steps:
                    time_step_update_slider.set_val(int(data_logger.num_time_steps))
                    break
                time_step_update_slider.set_val(int(val+self._play_speed))
                plt.pause(0.0000000001)

        update_drone_patterns(0)
        update_slider(0)
        self._play_speed = 1
        # Connect widgets with their respective update methods
        text_box_vmax.on_submit(update_heat_map_vmax)
        data_output_radio.on_clicked(update_output_plot)
        heat_map_radio.on_clicked(update_map_background_plot)
        drone_pattern_radio.on_clicked(drone_pattern_radio_update)
        speed_slider.on_changed(update_speed_box)
        pause_button.on_clicked(pause_button_update)
        start_button.on_clicked(play_button_update)
        time_step_update_slider.on_changed(update_slider)
        text_box.on_submit(update_text_box)
        back_button.on_clicked(back_button_update)
        next_button.on_clicked(next_button_update)

        if show:
            plt.show()
        return self._fig

    def update_plot(self, sim_model, time_step, data_logger):
        self.curr_index = time_step
        drone_positions = np.asarray(data_logger.drone_history[time_step])
        trash_positions = np.asarray(data_logger.trash_history[time_step])
        collector_positions = np.asarray(data_logger.collector_positions)
        charger_positions = np.asarray(data_logger.charger_positions)

        self._minute_time_text.set_text('%.2f'%(time_step/60) + " minutes")
        self._hour_time_text.set_text('%.2f'%(time_step/60/60) + " hours")
        self._drone_scatter.set_offsets(drone_positions)
        if self._show_trash_detection_radius_circle:
            if self._show_trash_detection_radius_circle:
                for circle in self.all_circles:
                    circle.remove()
                self.all_circles = []
            for drone_position in drone_positions:
                circle = plt.Circle((drone_position[0],drone_position[1]), self._trash_detection_radius,
     color='b', fill=False)
                self.all_circles.append(circle)
                self._main_ax.add_artist(circle)
        if self._drone_color_change_battery_level_on:
            battery_life = data_logger.drone_battery_life[time_step]
            battery_level_array = np.transpose(battery_life)
            n = mpl.colors.Normalize(vmin=-0.3, vmax =1)
            m = mpl.cm.ScalarMappable(norm=n, cmap='Greys')
            scat = self._drone_scatter
            scat.set_clim(vmin=-0.3, vmax=1)
            scat.set_facecolor(m.to_rgba(battery_level_array))

        self._collector_scatter.set_offsets(collector_positions)
        self._charger_scatter.set_offsets(charger_positions)
        if sim_model.persons_on:
            if len(person_positions) == 0:
                self._person_scatter.set_offsets(self.empty_array())
            else:
                self._person_scatter.set_offsets(person_positions)
        if len(trash_positions) == 0:
            self._trash_scatter.set_offsets(self.empty_array())
        else:
            self._trash_scatter.set_offsets(trash_positions)
        if self._all_max_trash_indices[time_step] == -1:
            self._single_longest_trash_scatter.set_offsets(self.empty_array())
        else:
            self._single_longest_trash_scatter.set_offsets(trash_positions[self._all_max_trash_indices[time_step]])

        self._pointing_arrow.remove()
        self._pointing_arrow = self._data_axis.arrow(time_step, 0, 0, self.data_y_max, width=0.1, length_includes_head=True)
        self._data_update_method(sim_model, time_step)
        self._heat_map_data_update_method(sim_model, time_step)

        self._fig.canvas.draw_idle()

    def close_plot(self):
        plt.close()

    def set_speed(self, speed):
        self._speed = speed
        return self

    def _clear_heat_maps(self):
        if self._cax is not None:
            self._cax.remove()
            self._cax = None
        if self._drone_colorbar is not None:
            # self._drone_colorbar.remove()
            self._drone_colorbar = None
        if self._drone_heatmap is not None:
```

```python
            self._drone_heatmap.remove()
            self._drone_heatmap = None
        if self._trash_colorbar is not None:
            # self._trash_colorbar.remove()
            self._trash_colorbar = None
        if self._trash_heatmap is not None:
            self._trash_heatmap.remove()
            self._trash_heatmap = None
        self._fig.canvas.draw_idle()

    # Unfortunately I had to duplicate the code with the heat maps in order to get them to clear and
 change properly
    def _plot_last_visited_step_heatmap(self, sim_model, time_step):
        self._clear_heat_maps()
        map_len = sim_model.park.bounds
        def update_trash_per_time_step(sim_model, time_step):
            heat_map = self._all_drone_heat_maps[time_step]
            self._drone_heatmap.set_data(heat_map.T)
        heat_map = self._all_drone_heat_maps[time_step]
        extent = (0,map_len,0,map_len)
        vmin = 0
        vmax = self._vmax
        self._drone_heatmap = self._main_ax.imshow(heat_map.T, vmin=vmin, vmax=vmax, interpolation='
 nearest', origin='lower', extent=extent)
        # Allocate space for the colorbar
        ax = self._main_ax
        self._cax = self._fig.add_axes([ax.get_position().x1-0.01, ax.get_position().y0, 0.01, ax.
 get_position().height])
        self._drone_colorbar = plt.colorbar(self._drone_heatmap, cax=self._cax)
        self._heat_map_data_update_method = update_trash_per_time_step
        if self._use_default_vmax:
            self._text_box_vmax.set_val(self._max_drone_heatmap)

    def _plot_weighted_trash_per_time_step_heatmap(self, sim_model, time_step):
        self._clear_heat_maps()
        map_len = sim_model.park.bounds
        def update_trash_per_time_step(sim_model, time_step):
            heat_map = self._all_trash_heat_maps[time_step]
            self._trash_heatmap.set_data(heat_map.T)
        heat_map = self._all_trash_heat_maps[time_step]
        extent = (0,map_len,0,map_len)
        self._trash_heatmap = self._main_ax.imshow(heat_map.T, vmin=0, vmax=self._vmax, cmap='Blues',
 interpolation='nearest', origin='lower', extent=extent)
        # Allocate space for the colorbar
        ax = self._main_ax
        self._cax = self._fig.add_axes([ax.get_position().x1-0.01, ax.get_position().y0, 0.01, ax.
 get_position().height])
        self._trash_colorbar = plt.colorbar(self._trash_heatmap, cax=self._cax)
        self._heat_map_data_update_method = update_trash_per_time_step
        if self._use_default_vmax:
            self._text_box_vmax.set_val(self._max_trash_heat_map)

    def _plot_number_of_times_visited(self, sim_model):
        self._clear_heat_maps()
        map_len = sim_model.park.bounds
        def update_trash_per_time_step(sim_model, time_step):
            pass
        heat_map = self._number_times_visited
        extent = (0,map_len,0,map_len)
        self._trash_heatmap = self._main_ax.imshow(heat_map.T, vmin=0, vmax=self._vmax, cmap='Blues',
    interpolation='nearest', origin='lower', extent=extent)
        # Allocate space for the colorbar
        ax = self._main_ax
        self._cax = self._fig.add_axes([ax.get_position().x1-0.01, ax.get_position().y0, 0.01, ax.
    get_position().height])
        self._trash_colorbar = plt.colorbar(self._trash_heatmap, cax=self._cax)
        self._heat_map_data_update_method = update_trash_per_time_step
        if self._use_default_vmax:
            self._text_box_vmax.set_val(self._max_number_times_visited)

    def _plot_average_heat_map_value(self, sim_model):
        self._clear_heat_maps()
        map_len = sim_model.park.bounds
        def update_trash_per_time_step(sim_model, time_step):
            pass
        heat_map = self._average_drone_heat_map
        extent = (0,map_len,0,map_len)
        self._trash_heatmap = self._main_ax.imshow(heat_map.T, vmin=0, vmax=self._vmax, cmap='Blues',
    interpolation='nearest', origin='lower', extent=extent)
        # Allocate space for the colorbar
        ax = self._main_ax
        self._cax = self._fig.add_axes([ax.get_position().x1-0.01, ax.get_position().y0, 0.01, ax.
    get_position().height])
        self._trash_colorbar = plt.colorbar(self._trash_heatmap, cax=self._cax)
        self._heat_map_data_update_method = update_trash_per_time_step
        if self._use_default_vmax:
            self._text_box_vmax.set_val(self._max_average_drone_heat_map)

    def _plot_num_trash_heat_map(self, sim_model):
        self._clear_heat_maps()
        map_len = sim_model.park.bounds
        def update_trash_per_time_step(sim_model, time_step):
            pass
        heat_map = self._num_trash_heat_map
        extent = (0,map_len,0,map_len)
        self._trash_heatmap = self._main_ax.imshow(heat_map.T, vmin=0, vmax=self._vmax, cmap='Blues',
 interpolation='nearest', origin='lower', extent=extent)
        # Allocate space for the colorbar
        ax = self._main_ax
        self._cax = self._fig.add_axes([ax.get_position().x1-0.01, ax.get_position().y0, 0.01, ax.
 get_position().height])
        self._trash_colorbar = plt.colorbar(self._trash_heatmap, cax=self._cax)
        self._heat_map_data_update_method = update_trash_per_time_step
        if self._use_default_vmax:
            self._text_box_vmax.set_val(self._max_num_trash_heat_map)

    def _plot_avg_since_last_visited_plot(self, ax, sim_model, data_logger):
        x = len(self._all_mean)
        y = self._all_mean
        self._static_data_plot(x, y, AVG_TIME_SINCE_VISITED, ax, sim_model, data_logger)

    def _plot_active_ratios_plot(self, ax, sim_model, data_logger):
        # TODO update this plot
        y1, y2, all_ratios = data_logger.active_drone_ratio()
        self._static_data_plot_multiple([all_ratios], ACTIVE_RATIO, ax, sim_model)

    def _plot_max_since_last_visited_plot(self, ax, sim_model, data_logger):
        x = len(self._all_max)
        y = self._all_max
        self._static_data_plot(x, y, MAX_TIME_SINCE_VISITED, ax, sim_model, data_logger)

    def _plot_std_dev_since_last_visited_plot(self, ax, sim_model, data_logger):
        x = len(self._all_std_dev)
        y = self._all_std_dev
        self._static_data_plot(x, y, STD_DEV_TIME_SINCE_VISITED, ax, sim_model, data_logger)

    def _plot_trash_per_time_step_plot(self, ax, sim_model, data_logger):
        x, trashes = data_logger.get_trash_per_time_step_data()
        self._static_data_plot(x, trashes, TRASH_PER_TIME_STEP_TITLE, ax, sim_model, data_logger)

    def _plot_avg_trash_left_out_in_each_time_step_plot(self, ax, sim_model, data_logger):
        x, trash_time = data_logger.get_running_avg_num_trash_per_timestep_data()
        self._static_data_plot(x, trash_time, AVG_TRASH_LEFT_OUT, ax, sim_model, data_logger)

    def _plot_max_time_left_out_in_each_time_step_plot(self, ax, sim_model, data_logger):
        x, max_time = data_logger.max_trash_left_out_each_time_step_data()
        self._static_data_plot(x, max_time, LONGEST_CURRENT_TRASH, ax, sim_model, data_logger)

    def _plot_avg_time_trash_left_out_in_each_time_step_plot(self, ax, sim_model, data_logger):
        x, trash_time = data_logger.avg_time_trash_left_out_in_each_time_step_data()
        self._static_data_plot(x, trash_time, AVG_TIME_TRASH_LEFT_OUT, ax, sim_model, data_logger)

    def _static_data_plot_multiple(self, y_datas, title, ax, sim_model, data_logger):
        ax.cla()
        x = len(sim_model.data_logger.trash_history)
        ax.set_xlim(0, x)
        max_values = []
        for y_data in y_datas:
            ax.plot(list(range(x)), y_data)
            max_values.append(max(y_data))
        max_value = max(max_values)
        if max_value == 0:
           max_value = 1
        ax.set_ylim(0, max_value)
        self.data_y_max = max_value
        ax.set_title(title)
        self._pointing_arrow = ax.arrow(self.curr_index, 0, 0, self.data_y_max, width=0.1,
    length_includes_head=True)
        def update_trash_per_time_step(sim_model, time_step):
            pass
        self._data_update_method = update_trash_per_time_step
        aspect = np.diff(self._data_axis.get_xlim()) / np.diff(self._data_axis.get_ylim())
        self._data_axis.set_aspect(aspect)

    def _static_data_plot(self, x_data, y_data, title, ax, sim_model, data_logger):
        ax.cla()
        x = len(data_logger.trash_history)
        ax.set_xlim(0, x)
        ax.plot(list(range(x)), y_data)
        max_value = max(y_data)
        if max_value == 0:
           max_value = 1
        ax.set_ylim(0, max_value)
        self.data_y_max = max_value
        ax.set_title(title)
        self._pointing_arrow = ax.arrow(self.curr_index, 0, 0, self.data_y_max, width=0.1,
    length_includes_head=True)
        def update_trash_per_time_step(sim_model, time_step):
            pass
        self._data_update_method = update_trash_per_time_step
        aspect = np.diff(self._data_axis.get_xlim()) / np.diff(self._data_axis.get_ylim())
        self._data_axis.set_aspect(aspect)


    def _plot_the_outputs(self, main_ax, sim_model, data_logger):
        x_place = 2.54
        y_place = 0.5
        output_dict = {}
        output_dict[MAX_TIME_LEFT_OUT] = data_logger.get_max_time_any_trash_left_out()
        output_dict[AVERAGE_TIME_TRASH_LEFT_OUT] = round(data_logger.get_avg_time_trash_left_out(),2)
        output_dict[AVG_NUM_TRASH_PER_TIMESTEP] = round(data_logger.get_avg_num_trash_in_sim(),2)
        times = data_logger.drones_with_depleted_energy_times
        if len(times) > 0:
            output_dict[RUN_OUT_BATTERY_TIMES] = times[0]
        output_dict[NUM_DRONES_TO_RUN_OUT_OF_BATTERIES] = data_logger.get_num_drones_ran_out_of_batteries
    ()
        output_dict[AVERAGE_TIME_SPENT_SEARCHING_PER_DRONE] = round(data_logger.
    get_avg_time_spent_searching_per_drone(),2)
        output_dict[AVERAGE_TIME_SPENT_COLLECTING_PER_DRONE] = round(data_logger.
    get_avg_time_spent_collecting_per_drone(),2)
        text = ""
        props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)
        for key, value in output_dict.items():
            if key in (AVERAGE_TIME_TRASH_LEFT_OUT):
                key = "Average time trash left out (s)"
            if key in (NUM_DRONES_TO_RUN_OUT_OF_BATTERIES):
                key = "# UAVs lost power"
            if key in (MAX_TIME_LEFT_OUT):
                key = "Max time any trash left out (s)"
            if key in (AVERAGE_TIME_SPENT_CHARGING_PER_DRONE):
                key = "Avg UAV charge time (s)"
            if key in (AVERAGE_TIME_SPENT_SEARCHING_PER_DRONE):
                key = "Avg UAV search time (s)"
            if key in (AVG_TIME_NOT_CHARGING_OR_SEARCHING):
                key = "Avg UAV in other states (s)"
            next_text = key + ": \n" + str(value) + "\n"
            text += next_text
        main_ax.text(x_place, y_place, text, transform=main_ax.transAxes, bbox=props)

    def _plot_the_inputs(self, main_ax):
        x_place = -0.47
        y_place = 0.35
        dont_print_these = (
            RANDOM_SEED,
            DRONE_SPEED,
            FOUND_DISTANCE,
            EMERGENCY_RECHARGE_LEVEL,
            SET_OUT_FOR_TRASH_WHILE_CHARGING_LEVEL,
            RETURN_TO_CHARGE_FROM_SEARCHING,
            FLY_TIME,
            RECHARGE_TIME,
            TRASH_PICKUP_DELAY,
            TRASH_DROPOFF_DELAY,
            INIT_CHARGERS_RANDOM,
            INIT_COLLECTORS_RANDOM,
            FAILED_EXPERIMENT,
            'Unnamed: 0'
        )
        text = ""
        props = dict(boxstyle='round', facecolor='wheat', alpha=0.5)
        for key, value in self._input_dict.items():
            if key in dont_print_these:
                continue
            if key in (NUMBER_OF_DRONES):
                key = "Number of UAVs"
            if key in (PARK_SIZE):
                key = "Park side length (m)"
            if key == SEARCH_PATTERN:
                value = SEARCH_PATTERNS[value]
            if key in (TRASH_GENERATION_RATE):
                # key = r'$\gamma_{T}$'
                key = "Trash Generation Rate"
                value *= 3600
                value = round(value, 2)
            if key in (TRASH_DETECTION_RADIUS):
                key = "Detection Distance (m)"
                value = round(value, 2)
            if key in (SIM_RUN_TIME):
                value = round(value, 2)
            next_text = key + ": \n" + str(value) + "\n"
            text += next_text
        main_ax.text(x_place, y_place, text, transform=main_ax.transAxes, bbox=props)

    @staticmethod
    def empty_array():
        return np.c_[np.array([]), np.array([])]

    def _plot_outside_bounds(self, side_length, ax):
        l = side_length
        bounds_x, bounds_y = [0, l, l, 0, 0],[0, 0, l, l, 0]
        polygon = np.column_stack((bounds_x,bounds_y))
        bounds = mplPath.Path(polygon)
        vertices = bounds.vertices
        ax.plot(vertices[:, 0], vertices[:, 1], color='violet')

    def _plot_park_paths(self, sim_model, ax):
        for node in sim_model.park.nodes:
            x_node = []
            y_node = []
            for child in node.children:
                x_node.append(node.coordinates[0])
                x_node.append(child.coordinates[0])
                y_node.append(node.coordinates[1])
                y_node.append(child.coordinates[1])
            ax.plot(x_node,y_node,color='deepskyblue', linewidth = 0.5)
```

A.1.3 Collector Placement Algorithm

coarse_genetic_fine_convex_optimization.py

```
import numpy as np
import scipy.optimize
from scipy.optimize import differential_evolution
from scipy.optimize import minimize
from scipy.optimize import Bounds
from scipy.optimize import NonlinearConstraint
from scipy.spatial import distance_matrix
from matplotlib import pyplot as plt
import time


def get_coords(discretization, map_min, map_max):
    x_dis = np.linspace(map_min, map_max, discretization)
    y_dis = np.linspace(map_min, map_max, discretization)
    return np.array([np.repeat(x_dis, discretization), np.tile(y_dis, discretization)]).T


def obj_maxmin(x, coordinates):
    collectors = np.array(x).reshape(-1, 2)
    distances = distance_matrix(coordinates, collectors, p=2)
    return np.max(np.min(distances, axis=1))


def obj_avgmin(x, coordinates):
    collectors = np.array(x).reshape(-1, 2)
    distances = distance_matrix(coordinates, collectors, p=2)
    return np.mean(np.min(distances, axis=1))


def scipy_differential(obj, lb, ub, number_collectors, coarse_coords):
    bounds_dif_ev = []
    for _ in range(number_collectors):
        bounds_dif_ev.append((lb, ub))
        bounds_dif_ev.append((lb, ub))
    return scipy.optimize.differential_evolution(obj, bounds_dif_ev, args=(coarse_coords,), maxiter
     =100000000)

# Convex minimization with fine grid
def minimize_results(start_x, obj, discretization, map_min, map_max):
    fine_coords = get_coords(discretization, map_min, map_max)
    return scipy_minimize(obj, start_x, fine_coords)


def scipy_minimize(obj, start_x, fine_coords):
    start_opt = time.time()
    result = minimize(obj, start_x, args=(fine_coords,))  # bounds=bounds_min)
    end_opt = time.time()
    time_taken = end_opt-start_opt
    print("nfev: {}, nit: {}, njev: {}, success: {}, time_taken: {}".format(
        result['nfev'], result["nit"], result['njev'], result['message'], time_taken))
    # Plot optimal configuration
    return result['x']


def run_design_coarse_GA_then_fine_scipy(lb, ub, map_min, map_max, number_collectors, obj, save=False,
     folder_name=""):
    # Global optimum finding algorithm with coarse grid
    start = time.time()
    coarse_coords = get_coords(30, map_min, map_max)
    result_coarse = scipy_differential(obj, lb, ub, number_collectors, coarse_coords)
    end = time.time()
    GA_time = end-start
    print(result_coarse)

    # Save results to compare
    start_x = result_coarse['x']
    start_x_pl = start_x.reshape(-1, 2)
    x = start_x

    x5 = minimize_results(x, obj, 500, map_min, map_max)
    final_x = x5
    print("Final x: {}".format(final_x))
    if save:
        if folder_name == "":
            folder_name = "test"
        np.savetxt("collector_placement_algorithms/data/{}/data{}.txt".format(
            folder_name, number_collectors), final_x)
    end = time.time()
    print("Total time {}: {}".format(number_collectors, end-start))
    print("GA_time {}: {}".format(number_collectors, GA_time))
    return start_x_pl, final_x

# Plot the two solutions to compare
def plot_solutions(start_x, x, map_min, map_max):
    x_pl = x.reshape(-1, 2)
    plt.scatter(x_pl[:, 0], x_pl[:, 1], label="After min")
    plt.scatter(start_x[:, 0], start_x[:, 1], label="After Genetic")
    plt.legend()

    plt.xlim(0, 100)
    plt.ylim(0, 100)
    plt.title("Optimal configuration")
```

```python

    # Plot distribution of distances
    plt.subplots()
    collectors = np.array(x).reshape(-1, 2)
    distances = distance_matrix(get_coords(500, map_min, map_max), collectors)
    d = np.min(distances, axis=1)

    number_collectors, bins, patches = plt.hist(x=d, bins='auto', color='#0504aa',
                                                alpha=0.7, rwidth=0.85)
    plt.grid(axis='y', alpha=0.75)
    plt.xlabel('Value')
    plt.ylabel('Frequency')
    plt.title('Histogram')
    # plt.text(23, 45, r'$\mu=, b=3$')
    maxfreq = number_collectors.max()
    # Set a clean upper y-axis limit.
    plt.ylim(ymax=np.ceil(maxfreq / 10) * 10 if maxfreq % 10 else maxfreq + 10)
    plt.show()

def run_experiments(folder_name):
    lb = 0
    ub = 100
    map_min = 0
    map_max = 100
    for number_collectors in range(1, 15):
        run_design_coarse_GA_then_fine_scipy(
            lb, ub, map_min, map_max, number_collectors, obj_maxmin, save=True, folder_name=folder_name)

def run_one_experiment():
    lb = 0
    ub = 100
    map_min = 0
    map_max = 100
    number_collectors = 5
    folder_name = "maxmin_test"
    start, final = run_design_coarse_GA_then_fine_scipy(lb, ub, map_min, map_max, number_collectors,
     obj_maxmin, save=True, folder_name=folder_name)
    plot_solutions(start, final, map_min, map_max)

if __name__ == "__main__":
    folder_name = "maxmin_overnight"
    run_experiments(folder_name)
```

A.1.4 DOE Generator

design_of_experiment_generator.py

```python
import os
import random

from pyDOE import lhs
import numpy as np
import pandas as pd

from experiment_runner.experiment_runner.path_manager import PathManager
RANDOM_SEED = "Random Seed"


class DesignOfExperimentGenerator():
    def __init__(self):
        self._inputs = {}
        self._repeated_experiments = False
        self._number_of_repeats = None
        self._random_seeds = None

    def add_input_with_range(self, name, min_value, max_value, is_int=False):
        self._inputs[name] = RangedInput(name, min_value, max_value, is_int)
        return self

    def add_constant_input(self, name, value, is_int=False):
        self._inputs[name] = ConstantInput(name, value, is_int)
        return self

    def add_leveled_input(self, name, levels):
        self._inputs[name] = LeveledInput(name, levels)
        return self

    def make_latin_hypercube_doe(self, number_of_experiments, save_to_csv, csv_name=None):
        count = self._count_non_constant_inputs()
        lhs_DOE = lhs(count, samples=number_of_experiments)
        return self._make_doe_helper(lhs_DOE, number_of_experiments, save_to_csv, csv_name)

    def make_monte_carlo_doe(self, number_of_experiments, save_to_csv, csv_name=None):
        count = self._count_non_constant_inputs()
        monte_carlo_DOE = np.random.rand(number_of_experiments, count)
        return self._make_doe_helper(monte_carlo_DOE, number_of_experiments, save_to_csv, csv_name)

    def _make_doe_helper(self, DOE, number_of_experiments, save_to_csv, csv_name=None):
        labeled_data = self._create_data_frame(DOE, number_of_experiments)
        final_DOE = pd.DataFrame(labeled_data)
        if save_to_csv:
            path = self._get_path(csv_name)
            final_DOE.to_csv(path)
        return final_DOE

    def with_repeated_experiments(self, number_of_experiments_per_data_point, random_seeds=None):
        # If this is activated, each experiment replicate set will get the same random seed
        self._repeated_experiments = True
        self._number_of_repeats = number_of_experiments_per_data_point
        if random_seeds == None:
            random_seeds = self._create_random_seeds_for_experiments(number_of_experiments_per_data_point
)
        else:
            if len(random_seeds) != number_of_experiments_per_data_point:
                raise Exception("There is a discrepancy in the number of \
                    experiments and the length of the random seed list")
        self._random_seeds = random_seeds
        return self

    def _create_random_seeds_for_experiments(self, number_of_experiments_per_data_point):
        random_seeds = []
        for _ in range(number_of_experiments_per_data_point):
            random_seeds.append(random.randrange(100000000))
        if len(random_seeds) > len(set(random_seeds)):
            raise Exception("Random seeds are not unique")
        return random_seeds

    def _get_path(self, csv_name):
        path = PathManager.input_doe_path()
        if not os.path.exists(path):
            os.makedirs(path)
        return PathManager.input_doe_csv_path(csv_name)

    def _count_non_constant_inputs(self):
        count = 0
        for input_field in self._inputs:
            if (not isinstance(self._inputs[input_field], ConstantInput)):
                count += 1
        return count

    def _create_data_frame(self, DOE, number_of_experiments):
        index = 0
        data_frame_dict = {}
        # Since the DOE input is normalized with values from zero to one,
        # we need to modify the values to be in the correct range with the
        # correct data type, add constant inputs and, if specified, add repeat
        # experiments
        for input_name in self._inputs:
            if (not isinstance(self._inputs[input_name], ConstantInput)):
                self._add_ranged_input_to_DOE(input_name, index, data_frame_dict, DOE)
                index += 1
            else:
                self._add_constant_input_to_DOE(input_name, data_frame_dict, number_of_experiments)
        if self._repeated_experiments:
            data_frame_dict = self._add_repeated_experiments(data_frame_dict, number_of_experiments)
        return data_frame_dict

    def _add_repeated_experiments(self, data_frame_dict, number_of_experiments):
        for input_name in self._inputs:
            data = data_frame_dict[input_name]
            to_add = np.array([])
            for index in range(self._number_of_repeats):
                to_add = np.append(to_add, data)
            input_field = self._inputs[input_name]
            if input_field.is_int:
                to_add = np.floor(to_add).astype(int)
            data_frame_dict[input_name] = to_add
        random_seed_data = np.array([])
        for index in range(self._number_of_repeats):
            random_seed_data = np.append(random_seed_data, np.full(number_of_experiments, self.
    _random_seeds[index]))
        data_frame_dict[RANDOM_SEED] = random_seed_data
        return data_frame_dict

    def _add_ranged_input_to_DOE(self, input_name, index, data_frame_dict, DOE):
        input_field = self._inputs[input_name]
        data = self._set_limits(
            DOE[:, index],
            input_field.min_value,
            input_field.max_value
            )
        if input_field.is_int:
            data = np.floor(data).astype(int)
        data_frame_dict[input_name] = data

    def _set_limits(self, column, lower_bound, upper_bound):
        column = column*(upper_bound-lower_bound)+lower_bound
        return column

    def _add_constant_input_to_DOE(self, input_name, data_frame_dict, number_of_experiments):
        constant_value = self._inputs[input_name].value
        data_frame_dict[input_name] = np.full(
            (number_of_experiments), constant_value)

class RangedInput():
    def __init__(self, name, min_value, max_value, is_int):
        self.name = name
        self.min_value = min_value
        if is_int:
            # Make the max value += 1 for ints
            # So that all the values get represented equally
            # by flooring the number in DOE creation
            max_value+=1
        self.max_value = max_value
        self.is_int = is_int


class ConstantInput():
    def __init__(self, name, value, is_int):
        self.name = name
        self.value = value
        self.is_int = is_int


class LeveledInput():
    def __init__(self, name, levels):
        self.name = name
        self.levels = levels
```

generate_doe_with_repeated_experiments.py

```
from doe_generator.doe_generator.design_of_experiment_generator import DesignOfExperimentGenerator
from experiment_runner.experiment_runner.parkcleanup_experiment_runner import ParkCleanupExperimentRunner

RANGED_PARAMETER_1 = "The first ranged parameter"
RANGED_PARAMETER_2 = "The second ranged parameter"
RANGED_PARAMETER_3 = "The third range parameter"
RANGED_PARAMETER_INT_1 = "The first ranged parameter that only contains ints"
RANGED_PARAMETER_INT_2 = "The second ranged parameter that only contains ints"
CONSTANT_PARAMETER_1 = "The first constant parameter"
CONSTANT_PARAMETER_2 = "The second constant parameter"


def main():
    DOE_generator = (
        DesignOfExperimentGenerator()
        .add_input_with_range(RANGED_PARAMETER_1, min_value=0, max_value=10)
        .add_input_with_range(RANGED_PARAMETER_2, min_value=-40, max_value=10000)
        .add_input_with_range(RANGED_PARAMETER_3, min_value=-42.3, max_value=76.93)
        .add_input_with_range(RANGED_PARAMETER_INT_1, min_value=-20, max_value=10, is_int=True)
        .add_input_with_range(RANGED_PARAMETER_INT_2, min_value=0, max_value=10, is_int=True)
        .add_constant_input(CONSTANT_PARAMETER_1, value=3)
        .add_constant_input(CONSTANT_PARAMETER_2, value=20345.56)
        .with_repeated_experiments(3, random_seeds=[2342305982, 23059802395, 340958405])
    )
    DOE_generator.make_latin_hypercube_doe(100, save_to_csv=True, csv_name="latin_hypercube_test")
    DOE_generator.make_monte_carlo_doe(5342, save_to_csv=True, csv_name="monte_carlo_test")



```

```python
if __name__ == "__main__":
    main()
```

A.1.5 Experiment Runner

abstract_experiment_runner.py

```python
import os
import time
import pickle
import random
from multiprocessing import Pool
from abc import ABC, abstractmethod

import pandas as pd
import numpy as np
import psutil

from experiment_runner.experiment_runner.path_manager import PathManager
from experiment_runner.experiment_runner.string_constants import INDEX


class AbstractExperimentRunner(ABC):
    def __init__(self, doe_name):
        self._checkpoint_printing = False
        self._how_often_to_checkpoint = None
        self._DOE = None
        self._pickled = None
        self._doe_name = doe_name
        self._checkpoint_csv_saving = False
        self._start = 0
        self._end = None
        self._make_error_file_printing_path()
        self._output_minimum_distance_data = False

    def _make_error_file_printing_path(self):
        self._error_path = PathManager.error_path(self._doe_name)
        if not os.path.exists(self._error_path):
            os.makedirs(self._error_path)

    def with_checkpoint_printing(self, how_often_to_checkpoint):
        self._checkpoint_printing = True
        self._how_often_to_checkpoint_print = how_often_to_checkpoint
        return self

    def with_csv_output_checkpointing(self, how_often_to_checkpoint):
        self._checkpoint_csv_saving = True
        self._how_often_to_checkpoint_to_csv = how_often_to_checkpoint
        return self

    def with_start_experiment(self, start):
        self._start = start
        return self

    def with_end_experiment(self, end):
        self._end = end
        return self

    @abstractmethod
    def run_one_from_dict(self, values):
        '''
        Implement this method in a new class to run a single instance of your experiment
        Values is a dictionary of input values.
        Return a dictionary with keys of strings that will be the column names and alphanumeric values
    that will be the row values
        for a csv table
        '''
        pass

    def run_all_from_object(self, DOE):
        values = []
        if self._end is None:
            self._end = len(DOE)
        for index in range(self._start, self._end):
            value = self.run_one_from_object(index, DOE=DOE)
            if self._checkpoint_printing and (index%self._how_often_to_checkpoint_print == 0):
                print("Run " + str(index) + " completed")
            values.append(value)
            if self._checkpoint_csv_saving and (index%self._how_often_to_checkpoint_to_csv == self.
    _how_often_to_checkpoint_to_csv-1):
                self._save_to_output_csv(values)
        self._save_to_output_csv(values)

    def run_all_from_object_with_multiprocessing(self, DOE, num_workers=None):
        if num_workers is None:
            cpu_count = psutil.cpu_count()
            # Default to use one less core than is available
            # so that the extra core can do system processes
            num_workers = cpu_count-1
        inputs = []
        if self._end is None:
            self._end = len(DOE)
        for index in range(self._start, self._end):
            inputs.append(self._get_experiment_dict_from_pandas(DOE, index))
        if self._checkpoint_csv_saving:
            self.multiprocessing_with_csv_checkpointing(inputs, num_workers)
        else:
            with Pool(num_workers) as p:
                values = p.map(self.run_one_from_dict, inputs)
                self._save_to_output_csv(values)

    def multiprocessing_with_csv_checkpointing(self, inputs, num_workers):
        how_often_checkpoint = self._how_often_to_checkpoint_to_csv
        keep_going = True
        i = 0
        all_values = []
        # Split the inputs into groups of number equal to how_often_checkpoint
        # and process each group one at a time with multiprocessing
        while(keep_going):
            if i+how_often_checkpoint < len(inputs):
                selected_inputs = inputs[i:i+how_often_checkpoint]
            else:
                selected_inputs = inputs[i:len(inputs)]
                keep_going = False
            with Pool(num_workers) as p:
                values = p.map(self.run_one_from_dict, selected_inputs)
                all_values.extend(values)
                self._save_to_output_csv(all_values)
                print("Run " + str(i+how_often_checkpoint) + " completed")
            i += how_often_checkpoint

    def _save_to_output_csv(self, values):
        df = pd.DataFrame(values)
        path = PathManager.output_path()
        if not os.path.exists(path):
            os.makedirs(path)
        df.to_csv(PathManager.output_path_from_csv_name(self._doe_name))

    def run_all_from_csv(self, csv_name, multiprocessing=False, num_workers=None):
        DOE = self.get_data_frame_from_csv_name(csv_name)
        if multiprocessing:
            self.run_all_from_object_with_multiprocessing(DOE, num_workers=num_workers)
        else:
            self.run_all_from_object(DOE)

    def run_one_from_csv(self, csv_name, index):
        DOE = self.get_data_frame_from_csv_name(csv_name)
        self.run_one_from_object(index, DOE=DOE)

    def get_data_frame_from_csv_name(self, csv_name):
        path = PathManager.input_doe_csv_path(csv_name)
        return pd.read_csv(path)

```

```python
    def run_one_from_object(self, index, DOE):
        values = self._get_experiment_dict_from_pandas(DOE, index)
        return self.run_one_from_dict(values)

    def _get_experiment_dict_from_pandas(self, DOE, index):
        dict_ = DOE.iloc[index].to_dict()
        # The pandas method converts all values to floats, and so
        # we need to check if they should be ints and convert them
        for key in dict_:
            if np.issubdtype(DOE[key], np.integer):
                dict_[key] = int(dict_[key])
        dict_[INDEX] = index
        return dict_
```

parkcleanup_experiment_runner.py

```python
import random
import sys
import os
from traceback import format_exc
import time
import pathlib

import numpy as np
from matplotlib import pyplot as plt

from parkcleanup.parkcleanup.simulation.park_cleanup_simulation import ParkCleanupSimulation
from parkcleanup.parkcleanup.dataloggers.sim_data_logger import SimDataLogger
from parkcleanup.parkcleanup.builders.drone_builder import DroneBuilder
from parkcleanup.parkcleanup.builders.sim_model_builder import SimModelBuilder
from parkcleanup.parkcleanup.model.agents.drone_state_type import DroneStateType
from parkcleanup.parkcleanup.tools.helper import mean, std_dev

from experiment_runner.experiment_runner.abstract_experiment_runner import AbstractExperimentRunner
from experiment_runner.experiment_runner.string_constants import *
from experiment_runner.experiment_runner.data_output_string_constants import *

from experiment_runner.experiment_runner.path_manager import PathManager
from preferences import PATH_STRING
import pprint

class ParkCleanupExperimentRunner(AbstractExperimentRunner):
    def __init__(self, doe_name):
        super().__init__(doe_name)
        self._folders_initialized = False

    def run_one_from_dict(self, values, return_sim=False, data_logger=None, base_path=None):
        if data_logger is None:
            data_logger = SimDataLogger(10, 75, True)
        if base_path is None:
            PathManager.BASE_PATH = pathlib.Path(PATH_STRING)
        else:
            PathManager.BASE_PATH = base_path
        # Set the random seed from the values, if not, create one and save it
        start_time = time.time()
        if RANDOM_SEED in values:
            random_seed = values[RANDOM_SEED]
        else:
            random_seed = random.randrange(sys.maxsize)
            values[RANDOM_SEED] = random_seed
        random.seed(random_seed)
        try:
            sim_model_builder = (
                SimModelBuilder()
                .set_park_bounds(values[PARK_SIZE])
                .set_random_trash_generation_on(values[TRASH_GENERATION_RATE])
            )
            if values[INIT_COLLECTORS_RANDOM]:
                sim_model_builder.init_collectors_random(values[NUMBER_OF_COLLECTORS])
            else:
                sim_model_builder.init_collectors_from_file(values[NUMBER_OF_COLLECTORS], values[PARK_SIZE])

            if values[INIT_CHARGERS_RANDOM]:
                sim_model_builder.init_rechargers_random(values[NUMBER_OF_CHARGERS])
            else:
                sim_model_builder.init_rechargers_from_file(values[NUMBER_OF_CHARGERS], values[PARK_SIZE])

            charging_coords = sim_model_builder._all_recharger_coords
            drone_builder = (
                DroneBuilder(values[PARK_SIZE])
                .set_speed(values[DRONE_SPEED])
                .set_fly_time(values[FLY_TIME])
                .set_recharge_time(values[RECHARGE_TIME])
                .set_trash_detection_radius(values[TRASH_DETECTION_RADIUS])
                .set_object_found_distance(values[FOUND_DISTANCE])
                .set_constant_trash_dropoff_delay(values[TRASH_DROPOFF_DELAY])
                .set_constant_trash_pickup_delay(values[TRASH_PICKUP_DELAY])
                .set_charging_params(
                    set_out_for_seen_trash_while_charging=values[SET_OUT_FOR_TRASH_WHILE_CHARGING_LEVEL],
                    emergency_recharge_level=values[EMERGENCY_RECHARGE_LEVEL],
                    return_to_charge_from_patrolling=values[RETURN_TO_CHARGE_FROM_SEARCHING]
                )
                .set_number_of_drones_to_init(values[NUMBER_OF_DRONES])
                .set_starting_position_on_coordinates(charging_coords)
                .set_start_delay()
```

```python
            )
            if values[SEARCH_PATTERN] == 0:
                drone_builder.set_search_method_random_bounce()
            elif values[SEARCH_PATTERN] == 1:
                drone_builder.set_search_method_global_lawnmower()
            elif values[SEARCH_PATTERN] == 2:
                drone_builder.set_search_method_partitioned_random_bounce()
            else:
                drone_builder.set_search_method_partitioned_lawnmower()
            drones = drone_builder.commit()

            sim_model_builder.init_drones(drones)
            sim_model = sim_model_builder.commit()
            sim = ParkCleanupSimulation(sim_model)
            sim.run_sim(values[LENGTH_OF_SIMULATION], seed_for_run=random_seed, data_logger=data_logger)
        except:
            # Output any errors to an external file so that it doesn't break if you are running a set of experiments
            self._write_error_to_file(values[INDEX])
            # Return dictionary with minimal information for experiment identification
            values[FAILED_EXPERIMENT] = 1
            return values
        end_time = time.time()
        values[SIM_RUN_TIME] = end_time - start_time
        values[FAILED_EXPERIMENT] = 0
        if return_sim:
            return sim
        else:
            try:
                index = values[INDEX]
                bounds = values[PARK_SIZE]
                tdr = values[TRASH_DETECTION_RADIUS]
                return self._record_output_data(sim, values, index, bounds, tdr)
            except:
                # Output any errors to an external file so that it doesn't break if you are running a set of experiments
                self._write_error_to_file(values[INDEX])
                # Return dictionary with minimal information for experiment identification
                values[FAILED_EXPERIMENT] = 1
                return values

    def run_one_from_csv_with_plotting(self, csv_name, index, plotter, return_sim=False, values_to_replace=None, data_logger=None):
        DOE = self.get_data_frame_from_csv_name(csv_name)
        values = self._get_experiment_dict_from_pandas(DOE, index)
        if values_to_replace is not None:
            for key, pair in values_to_replace.items():
                values[key] = pair
        if return_sim:
            return self.run_one_from_dict(values, return_sim=True, data_logger=data_logger)
        else:
            plotter.show_inputs(values)
            start = time.time()
            sim = self.run_one_from_dict(values, return_sim=True, data_logger=data_logger)
            end = time.time()
            print(end-start)
            plotter.interactive_plot_data(sim)

    def test_experiment_outputs(self, csv_name, index, values_to_replace=None, data_logger=None):
        DOE = self.get_data_frame_from_csv_name(csv_name)
        values = self._get_experiment_dict_from_pandas(DOE, index)
        if values_to_replace is not None:
            for key, pair in values_to_replace.items():
                values[key] = pair
        return self.run_one_from_dict(values, return_sim=False, data_logger=data_logger)

    def _write_error_to_file(self, index):
        path = os.path.join(self._error_path, "run" + str(index))
        with open(path, 'w+') as f:
            f.write(format_exc())

    def _save_line_plot(self, x, y, title, index):
        fig = plt.figure()
        plt.plot(x, y)
        plt.xlim(0, max(x))
        plt.ylim(0, max(y))
        # plt.title(title)
        plt.savefig(PathManager.plot_save_output_path(self._doe_name, title, index))
        plt.close(fig=fig)

    def _save_charger_collector_plot(self, charger, collector, bounds, title, index):
        fig = plt.figure()
        plt.scatter(charger[:,0], charger[:,1], marker="P", color="m", label="Chargers")
        plt.scatter(collector[:,0], collector[:,1],  marker=r'$\sqcup$', color="saddlebrown", label="
    Collectors")
        plt.xlim(0, bounds)
        plt.ylim(0, bounds)
        plt.legend()
        plt.savefig(PathManager.plot_save_output_path(self._doe_name, title, index))
        plt.close(fig=fig)

    def _save_charger_plot(self, charger, bounds, title, index):
        fig = plt.figure()
        plt.scatter(charger[:,0], charger[:,1], marker="P", color="m", label="Chargers")
        plt.xlim(0, bounds)
        plt.ylim(0, bounds)
        # plt.title(title)
        plt.savefig(PathManager.plot_save_output_path(self._doe_name, title, index))
        plt.close(fig=fig)

    def _save_collector_plot(self, collector, bounds, title, index):
        fig = plt.figure()
        plt.scatter(collector[:,0], collector[:,1],  marker=r'$\sqcup$', color="saddlebrown", label="
     Collectors")
        plt.xlim(0, bounds)
        plt.ylim(0, bounds)
        # plt.title(title)
        plt.savefig(PathManager.plot_save_output_path(self._doe_name, title, index))
        plt.close(fig=fig)

    def _save_data(self, y, title, index):
        np.savetxt(PathManager.data_save_output_path(self._doe_name, title, index), y)

    def _save_heatmap(self, heat_map, bounds, title, index):
        fig, ax = plt.subplots()
        extent = (0,bounds,0,bounds)
        hm = ax.imshow(heat_map.T, vmin=0, vmax=np.max(heat_map), interpolation='nearest', origin='lower'
     , extent=extent)
        # ax.set_title(title)
        plt.colorbar(hm)
        plt.savefig(PathManager.plot_save_output_path(self._doe_name, title, index))
        plt.close(fig=fig)

    def _save_polys_and_lawnmower_plot(self, sim_model, bounds, index, title):
        there_are_polys = sim_model.all_drones[0].poly_of_area is not None
        there_are_patrols = sim_model.all_drones[0].patrol_coordinates is not None
        if there_are_polys or there_are_patrols:
            group_id = 0
            polys_to_plot = []
            patrols_to_plot = []
            for drone in sim_model.all_drones:
                if drone.group_index != group_id:
                    # Save stuff
                    fig = plt.figure()
                    if there_are_patrols:
                        self._save_coords(patrols_to_plot)
                    if there_are_polys:
                        self._save_partitions(polys_to_plot)
                    self._save_fig(fig, title, group_id, bounds, index)
                    polys_to_plot = []
                    patrols_to_plot = []
                    group_id += 1
                if there_are_polys:
                    polys_to_plot.append(drone.poly_of_area)
                if there_are_patrols:
                    patrols_to_plot.append(drone.patrol_coordinates)
            fig = plt.figure()
            if there_are_patrols:
                self._save_coords(patrols_to_plot)
            if there_are_polys:
                self._save_partitions(polys_to_plot)
            self._save_fig(fig, title, group_id, bounds, index)

    def _save_fig(self, fig, title, group_id, bounds, index):
        plt.title(title + " for Group{}".format(group_id))
        plt.xlim(0,bounds)
        plt.ylim(0,bounds)
        plt.savefig(PathManager.plot_save_output_path_with_groups(self._doe_name, title, index, group_id)
)
        plt.close(fig=fig)

    def _save_polys_and_coords(self, polys, coords):
        self._save_partitions(polys)
        self._save_coords(coords)

    def _save_partitions(self, polys):
        for poly in polys:
            plt.plot(*poly.exterior.xy, c='k')

    def _save_coords(self, coords_set):
        for coord in coords_set:
            coord = np.asarray(coord)
            plt.plot(coord[:,0], coord[:,1], c='b')

    def _record_output_data(self, simulation, csv_row_values, index, bounds, tdr):
        start_time = time.time()
        sim_model = simulation.sim_model
        data_logger = simulation.data_logger

        # This check saves time in a multirun experiment, so once the folders are initialized the
        # next runs will not check if the folders are there
        if not self._folders_initialized:
            PathManager.make_plot_save_output_path_folder(self._doe_name,
 STD_DEV_TIME_SINCE_SEARCHED_LINE_CHART)
            PathManager.make_plot_save_output_path_folder(self._doe_name,
 MAX_TIME_SINCE_SEARCHED_LINE_CHART)
            PathManager.make_plot_save_output_path_folder(self._doe_name,
 AVG_TIME_SINCE_SEARCHED_LINE_CHART)
            PathManager.make_plot_save_output_path_folder(self._doe_name, TRASH_PER_TIME_STEP_LINE_CHART)
            PathManager.make_plot_save_output_path_folder(self._doe_name, AVG_TRASH_LEFT_OUT_LINE_CHART)
            PathManager.make_plot_save_output_path_folder(self._doe_name,
 LONGEST_CURRENT_TRASH_LINE_CHART)
```

```
            PathManager.make_plot_save_output_path_folder(self._doe_name,
AVG_TIME_TRASH_LEFT_OUT_LINE_CHART)
            PathManager.make_plot_save_output_path_folder(self._doe_name, TOTAL_TRASH_TIME_LINE_CHART)
            PathManager.make_plot_save_output_path_folder(self._doe_name, NUMBER_TIMES_VISITED_HM)
            PathManager.make_plot_save_output_path_folder(self._doe_name, AVERAGE_TIME_LAST_SEARCHED_HM)
            PathManager.make_plot_save_output_path_folder(self._doe_name, NUM_TOTAL_TRASH_HM)
            PathManager.make_plot_save_output_path_folder(self._doe_name, AVG_TRASH_TIME_EACH_CELL_HM)
            PathManager.make_plot_save_output_path_folder(self._doe_name, CHARGER_LOCATIONS)
            PathManager.make_plot_save_output_path_folder(self._doe_name, COLLECTOR_LOCATIONS)
            PathManager.make_values_folder(self._doe_name, TRASH_INFO)
            PathManager.make_plot_folder(self._doe_name, CHARGER_AND_COLLECTOR_LOCATIONS)
            PathManager.make_plot_folder(self._doe_name, PARTITIONS_PATTERNS)
            self._folders_initialized = True

        x, trash_per_time_step = data_logger.get_trash_per_time_step_data()
        x, avg_trash_left_out = data_logger.get_running_avg_num_trash_per_timestep_data()
        x, longest_curr_trash = data_logger.max_trash_left_out_each_time_step_data()
        x, avg_time_trash_left_out = data_logger.avg_time_trash_left_out_in_each_time_step_data()
        x, total_trash_time = data_logger.get_total_trash_time_per_time_step_data()
        num_trash_heat_map = data_logger.num_trash_collected_heat_map
        avg_time_trash_heat_map = data_logger.times_left_out_heat_map
        num_times_visited = data_logger.get_num_times_visited_hm()
        avg_heat_map = data_logger.get_average_heat_map()

        trash_info = data_logger.all_trash_info
        all_max = data_logger.all_max_hm
        all_mean = data_logger.all_mean_hm
        all_std = data_logger.all_std_dev_hm
        # TODO plot std dev heat map
        self._save_line_plot(x, all_std, STD_DEV_TIME_SINCE_SEARCHED_LINE_CHART, index)
        self._save_line_plot(x, all_max, MAX_TIME_SINCE_SEARCHED_LINE_CHART, index)
        self._save_line_plot(x, all_mean, AVG_TIME_SINCE_SEARCHED_LINE_CHART, index)
        self._save_line_plot(x, trash_per_time_step, TRASH_PER_TIME_STEP_LINE_CHART, index)
        self._save_line_plot(x, avg_trash_left_out, AVG_TRASH_LEFT_OUT_LINE_CHART, index)
        self._save_line_plot(x, avg_time_trash_left_out, AVG_TIME_TRASH_LEFT_OUT_LINE_CHART, index)
        self._save_line_plot(x, longest_curr_trash, LONGEST_CURRENT_TRASH_LINE_CHART, index)
        self._save_line_plot(x, total_trash_time, TOTAL_TRASH_TIME_LINE_CHART, index)


        self._save_heatmap(num_times_visited, bounds, NUMBER_TIMES_VISITED_HM, index)
        self._save_heatmap(avg_heat_map, bounds, AVERAGE_TIME_LAST_SEARCHED_HM, index)
        self._save_heatmap(num_trash_heat_map, bounds, NUM_TOTAL_TRASH_HM, index)
        self._save_heatmap(avg_time_trash_heat_map, bounds, AVG_TRASH_TIME_EACH_CELL_HM, index)

        collector_coords = np.asarray([collector.position for collector in sim_model.all_collectors])
        charger_coords = np.asarray([charger.position for charger in sim_model.all_rechargers])
        self._save_charger_collector_plot(charger_coords, collector_coords, bounds,
CHARGER_AND_COLLECTOR_LOCATIONS, index)
```

```
        self._save_charger_plot(charger_coords, bounds, CHARGER_LOCATIONS, index)
        self._save_collector_plot(collector_coords, bounds, COLLECTOR_LOCATIONS, index)

        self._save_data(np.array(trash_info), TRASH_INFO, index)
        self._save_data(charger_coords, CHARGER_LOCATIONS, index)
        self._save_data(collector_coords, COLLECTOR_LOCATIONS, index)
        self._save_data(all_std, STD_DEV_TIME_SINCE_SEARCHED_LINE_CHART, index)
        self._save_data(all_max, MAX_TIME_SINCE_SEARCHED_LINE_CHART, index)
        self._save_data(all_mean, AVG_TIME_SINCE_SEARCHED_LINE_CHART, index)
        self._save_data(trash_per_time_step, TRASH_PER_TIME_STEP_LINE_CHART, index)
        self._save_data(avg_trash_left_out, AVG_TRASH_LEFT_OUT_LINE_CHART, index)
        self._save_data(avg_time_trash_left_out, AVG_TIME_TRASH_LEFT_OUT_LINE_CHART, index)
        self._save_data(longest_curr_trash, LONGEST_CURRENT_TRASH_LINE_CHART, index)
        self._save_data(total_trash_time, TOTAL_TRASH_TIME_LINE_CHART, index)

        self._save_data(num_times_visited, NUMBER_TIMES_VISITED_HM, index)
        self._save_data(avg_heat_map, AVERAGE_TIME_LAST_SEARCHED_HM, index)
        self._save_data(num_trash_heat_map, NUM_TOTAL_TRASH_HM, index)
        self._save_data(avg_time_trash_heat_map, AVG_TRASH_TIME_EACH_CELL_HM, index)

        self._save_polys_and_lawnmower_plot(sim_model, bounds, index, PARTITIONS_PATTERNS)

        csv_row_values[AVERAGE_VISIT_TIME] = all_mean[-1]
        csv_row_values[STD_DEV_VISIT_TIME] = np.std(avg_heat_map)

        csv_row_values[TOTAL_TRASH_COLLECTED] = data_logger.get_total_trash_picked_up()
        csv_row_values[TOTAL_TRASH_LEFT_OUT] = data_logger.get_total_number_of_unique_trash_in_sim()
        csv_row_values[AVERAGE_TIME_TRASH_LEFT_OUT] = data_logger.get_avg_time_trash_left_out()
        csv_row_values[AVERAGE_TIME_COLLECTED] = data_logger.get_avg_time_trash_collected()
        # Welches algorithm for std deviation needs to be implemented for this to work
        #csv_row_values[STD_DEV_TIME_TRASH_LEFT_OUT] = data_logger.get_std_dev_time_trash_left_out()
        csv_row_values[MAX_TIME_LEFT_OUT] = data_logger.get_max_time_any_trash_left_out()
        csv_row_values[AVG_NUM_TRASH_PER_TIMESTEP] = data_logger.get_avg_num_trash_in_sim()
        csv_row_values[MAX_NUM_TRASH_PER_TIMESTEP] = data_logger.get_max_num_trash_in_sim_any_time()

        csv_row_values[AVERAGE_TIME_SPENT_SEARCHING_PER_DRONE] = data_logger.
    get_avg_time_spent_searching_per_drone()
        csv_row_values[AVERAGE_TIME_SPENT_COLLECTING_PER_DRONE] = data_logger.
    get_avg_time_spent_collecting_per_drone()
        csv_row_values[NUM_DRONES_TO_RUN_OUT_OF_BATTERIES] = data_logger.
    get_num_drones_ran_out_of_batteries()
        end_time = time.time()
        csv_row_values[POSTPROCESS_TIME] = end_time - start_time
        csv_row_values[TOTAL_RUN_TIME] = csv_row_values[POSTPROCESS_TIME] + csv_row_values[SIM_RUN_TIME]
        return csv_row_values
```

data_output_string_constants.py

```
# String constants for data output folders
```

```
STD_DEV_TIME_SINCE_SEARCHED_LINE_CHART = "Snapshot of std dev of TLS HM cell values at each time step"
MAX_TIME_SINCE_SEARCHED_LINE_CHART = "Max TLS HM cell value at each time step"
AVG_TIME_SINCE_SEARCHED_LINE_CHART = "Snapshot of avg TLS HM cell values at each time step"
TRASH_PER_TIME_STEP_LINE_CHART = "Number trash at each time step"
AVG_TRASH_LEFT_OUT_LINE_CHART = "Running avg of number of trash left at each time step"
LONGEST_CURRENT_TRASH_LINE_CHART = "Left out value of the trash thats been out the longest at each time
    step"
AVG_TIME_TRASH_LEFT_OUT_LINE_CHART = "Running avg of time trash left out line chart (s)"
TOTAL_TRASH_TIME_LINE_CHART = "Total time left out of trash at each time step"

NUMBER_TIMES_VISITED_HM = "Overall number of times searched in each cell HM"
AVERAGE_TIME_LAST_SEARCHED_HM = "Overall avg TLS HM"
NUM_TOTAL_TRASH_HM = "Overall num trash in each cell HM"
AVG_TRASH_TIME_EACH_CELL_HM = "Overall avg time trash left out in each cell HM"
TRASH_INFO = "index, time appeared, time left out, position of each trash"
```

string_constants.py

```
# Inputs
NUMBER_OF_COLLECTORS = "Number of Collectors"
NUMBER_OF_CHARGERS = "Number of Chargers"
NUMBER_OF_DRONES = "Number of Drones"
DRONE_SPEED = "Drone Speed"
FOUND_DISTANCE = "Found Distance"
TRASH_DETECTION_RADIUS = "Trash Detection Radius"
EMERGENCY_RECHARGE_LEVEL = "Emergency Recharge Level"
SET_OUT_FOR_TRASH_WHILE_CHARGING_LEVEL = "Set out for Trash while Charging Level"
RETURN_TO_CHARGE_FROM_SEARCHING = "Return to Charge from Searching Level"
FLY_TIME = "Fly Time"
RECHARGE_TIME = "Recharge Time"
TRASH_PICKUP_DELAY = "Trash Pickup Delay"
TRASH_DROPOFF_DELAY = "Trash Dropoff Delay"
PARK_SIZE = "Park Size"
TRASH_GENERATION_RATE = "Trash Generation Rate"
LENGTH_OF_SIMULATION = "Length of Simulation"
RANDOM_SEED = "Random Seed"
INDEX = "Index"
SEARCH_PATTERN = "Search Pattern"
INIT_CHARGERS_RANDOM = "Init chargers random"
INIT_COLLECTORS_RANDOM = "Init collectors random"
RANDOM_BOUNCE = "Random Bounce"
GLOBAL_LAWNMOWER = "Global Lawnmower"
PARTITIONED_BOUNCE = "Partitioned bounce"
PARTITIONED_LAWNMOWER = "Partitioned lawnmower"
SEARCH_PATTERNS = (RANDOM_BOUNCE, GLOBAL_LAWNMOWER, PARTITIONED_BOUNCE, PARTITIONED_LAWNMOWER)

# Global Outputs
SIM_RUN_TIME = "Sim Run Time"
POSTPROCESS_TIME = "Postprocessing Time"
```

```
TOTAL_RUN_TIME = "Total Run Time"
TOTAL_TRASH_COLLECTED = "Total trash collected"
TOTAL_TRASH_LEFT_OUT = "Total trash left out"
AVERAGE_TIME_TRASH_LEFT_OUT = "Average time trash left out"
STD_DEV_TIME_TRASH_LEFT_OUT = "Standard deviation time trash left out"
AVERAGE_TIME_SPENT_SEARCHING_PER_DRONE = "Average time spent searching per drone"
STD_DEV_TIME_SPENT_SEARCHING_PER_DRONE = "Std deviation of time spent searching per drone"
AVERAGE_TIME_SPENT_CHARGING_PER_DRONE = "Average time spent charging per drone"
STD_DEV_TIME_SPENT_CHARGING_PER_DRONE = "Std deviation of time spent charging per drone"
AVERAGE_TIME_SPENT_COLLECTING_PER_DRONE = "Avg UAV collect time (s)"
STD_DEV_TIME_SPENT_COLLECTING_PER_DRONE = "Std deviation of time spent collecting per drone"

TOTAL_ENERGY_USED = "Total energy used"
AVG_ENERGY_USED_PER_DRONE = "Average energy used per drone"
STD_DEV_ENERGY_USED_PER_DRONE = "Std deviation of energy used per drone"
NUM_DRONES_TO_RUN_OUT_OF_BATTERIES = "Number of drones to run out of batteries"
RUN_OUT_BATTERY_TIMES = "Times Ran Out of Batteries"
AVG_TIME_SPENT_GOING_TO_TRASH = "Average time spent going to trash"
STD_DEV_TIME_SPENT_GOING_TO_TRASH = "Std deviation time spent going to trash"
MAX_TIME_LEFT_OUT = "Max time any trash was left out"
AVG_NUM_TRASH_PER_TIMESTEP = "Avg num trash per timestep"
MAX_NUM_TRASH_PER_TIMESTEP = "Max num trash per timestep"
AVG_TIME_NOT_CHARGING_OR_SEARCHING = "Avg time not charging or searching"
AVERAGE_VISIT_TIME = "Avg visit time to each cell"
STD_DEV_VISIT_TIME = "Std dev visit time to each cell"
AVG_TRASH_TIME_EACH_CELL = "Average time trash in each cell"
AVERAGE_TIME_COLLECTED = "Avg time to get to trash after appeared"

STD_DEV_CHARGER_USAGE = "Std deviation charger usage"
STD_DEV_COLLECTOR_USAGE = "Std deviation collector usage"
CENTROID_COLLECTOR_X = "X Centroid Collectors"
CENTROID_COLLECTOR_Y = "Y Centroid Collectors"
STD_DEV_COLLECTOR_X = "X Std Dev Collectors"
STD_DEV_COLLECTOR_Y = "Y Std Dev Collectors"

CENTROID_CHARGER_X = "X Centroid Chargers"
CENTROID_CHARGER_Y = "Y Centroid Chargers"
STD_DEV_CHARGERS_X = "X Std Dev Chargers"
STD_DEV_CHARGERS_Y = "Y Std Dev Chargers"

FAILED_EXPERIMENT = "Failed Experiment"

# Plot titles
TRASH_PER_TIME_STEP_TITLE = "Trash in simulation at each time step"
MAX_TIME_SINCE_VISITED = "Max time last visited"
AVG_TIME_SINCE_VISITED = "Avg time last visited"
TOTAL_TRASH = "Total trash"
LONGEST_CURRENT_TRASH = "Longest curr trash"
```

```python
AVG_TIME_TRASH_LEFT_OUT = "Avg time trash left out"
AVG_TRASH_LEFT_OUT = "Avg trash left out"
NUMBER_TIMES_VISITED = "# times visited"
AVERAGE_VISITED = "Avg visit time"
NUM_TOTAL_TRASH = "# Trash"
STD_DEV_TIME_SINCE_VISITED = "Std dev last visit"
ACTIVE_RATIO = "Active/Searching Drones"

# Data output names
CHARGER_AND_COLLECTOR_LOCATIONS = "Charger and collector Locations"
CHARGER_LOCATIONS = "Charger Locations"
COLLECTOR_LOCATIONS = "Collector Locations"
PARTITIONS_PATTERNS = "Partitions and or Patrol Patterns"
```

path_manager.py

```python
import pathlib

DATA_FOLDER = 'data'
INPUT_FOLDER = 'input'
OUTPUT_FOLDER = 'output'
RUN = 'run'
ERRORS_FOLDER = 'errors'

class PathManager:
    BASE_PATH = pathlib.Path.cwd()

    @staticmethod
    def input_doe_path():
        return PathManager.BASE_PATH / DATA_FOLDER / 'inputs'

    @staticmethod
    def input_doe_csv_path(csv_name):
        return PathManager.BASE_PATH / DATA_FOLDER / 'inputs' / (csv_name+'.csv')

    @staticmethod
    def error_path(name):
        return PathManager.BASE_PATH / DATA_FOLDER / 'errors' / name

    @staticmethod
    def output_path():
        return PathManager.BASE_PATH / DATA_FOLDER / 'output'

    @staticmethod
    def output_path_from_csv_name(name):
        return PathManager.BASE_PATH / DATA_FOLDER / 'output' / (name + ".csv")

    @staticmethod
    def make_plot_folder(csv_name, name):
        path = PathManager.get_plot_folder(csv_name, name)
        pathlib.Path.mkdir(path, parents=True, exist_ok=True)

    @staticmethod
    def make_values_folder(csv_name, name):
        path = PathManager.get_values_folder(csv_name, name)
        pathlib.Path.mkdir(path, parents=True, exist_ok=True)

    @staticmethod
    def get_plot_folder(csv_name, name):
        return PathManager.BASE_PATH / csv_name / 'plots' / name

    @staticmethod
    def get_values_folder(csv_name, name):
        return PathManager.BASE_PATH / csv_name / 'values' / name

    @staticmethod
    def make_plot_save_output_path_folder(csv_name, name):
        PathManager.make_plot_folder(csv_name, name)
        PathManager.make_values_folder(csv_name, name)

    @staticmethod
    def plot_save_output_path(csv_name, name, index):
        return PathManager.get_plot_folder(csv_name, name) / 'run{}'.format(index)

    @staticmethod
    def plot_save_output_path_with_groups(csv_name, name, index, group):
        return PathManager.get_plot_folder(csv_name, name) / 'run{}_group{}'.format(index, group)

    @staticmethod
    def data_save_output_path(csv_name, name, index):
        return PathManager.get_values_folder(csv_name, name) / 'run{}.txt'.format(index)
```

make_and_run_doe_with_multiprocessing.py

```
from doe_generator.doe_generator.design_of_experiment_generator import DesignOfExperimentGenerator
from experiment_runner.experiment_runner.parkcleanup_experiment_runner import ParkCleanupExperimentRunner
from experiment_runner.experiment_runner.string_constants import *
import time


def main():
    lhs_DOE = _make_DOE()
    start = time.time()
    experiment_runner = (
        ParkCleanupExperimentRunner('latin_hypercube_test')
    )
    experiment_runner.run_all_from_object_with_multiprocessing(lhs_DOE)
    end = time.time()
    print(end-start)

def _make_DOE():
    DOE_generator = (
        DesignOfExperimentGenerator()
        .add_constant_input(DRONE_SPEED, 3)
        .add_constant_input(FOUND_DISTANCE, 3)
        .add_constant_input(EMERGENCY_RECHARGE_LEVEL, 0.1)
        .add_constant_input(SET_OUT_FOR_TRASH_WHILE_CHARGING_LEVEL, 1.0)
        .add_constant_input(RETURN_TO_CHARGE_FROM_SEARCHING, 0.1)
        .add_constant_input(FLY_TIME, 1800, is_int=True)
        .add_constant_input(RECHARGE_TIME, 3600, is_int=True)
        .add_constant_input(TRASH_PICKUP_DELAY, 5, is_int=True)
        .add_constant_input(TRASH_DROPOFF_DELAY, 5, is_int=True)
        .add_constant_input(LENGTH_OF_SIMULATION, 42000, is_int=True)
        .add_constant_input(INIT_COLLECTORS_RANDOM, 0, is_int=True)
        .add_constant_input(INIT_CHARGERS_RANDOM, 0, is_int=True)
        .add_constant_input(SEARCH_PATTERN, 3, is_int=True)
        .add_input_with_range(NUMBER_OF_COLLECTORS, 1, 10, is_int=True)
        .add_input_with_range(NUMBER_OF_CHARGERS, 1, 10, is_int=True)
        .add_input_with_range(NUMBER_OF_DRONES, 3, 27, is_int=True)
        .add_input_with_range(PARK_SIZE, 200, 800, is_int=True)
        .add_input_with_range(TRASH_DETECTION_RADIUS, 10, 50)
        .add_input_with_range(TRASH_GENERATION_RATE, 0.003, 0.03)
    )
    lhs_DOE = DOE_generator.make_latin_hypercube_doe(6, save_to_csv=True, csv_name='latin_hypercube_test'
    )
    return lhs_DOE


if __name__ == "__main__":
    main()
```

A.2 Chapter 2 Code

four_strategy_framework_paper.py

```
from doe_generator.doe_generator.design_of_experiment_generator import DesignOfExperimentGenerator
from experiment_runner.experiment_runner.parkcleanup_experiment_runner import ParkCleanupExperimentRunner
from experiment_runner.experiment_runner.string_constants import *
from experiment_runner.experiment_runner.path_manager import PathManager

import time
import pathlib
from preferences import PATH_STRING

EXPERIMENT_NAME = 'four_strategy_march20'
```

```python
def main():
    PathManager.BASE_PATH = pathlib.Path(PATH_STRING)
    #lhs_DOE = _make_DOE()
    start = time.time()
    experiment_runner = (
        ParkCleanupExperimentRunner(EXPERIMENT_NAME)
        .with_csv_output_checkpointing(50)
    )
    experiment_runner.run_all_from_csv(EXPERIMENT_NAME, multiprocessing=True, num_workers=3)
    end = time.time()
    print(end-start)

def _make_DOE():
    DOE_generator = (
        DesignOfExperimentGenerator()
        .add_constant_input(DRONE_SPEED, 3)
        .add_constant_input(FOUND_DISTANCE, 3)
        .add_constant_input(EMERGENCY_RECHARGE_LEVEL, 0.1)
        .add_constant_input(SET_OUT_FOR_TRASH_WHILE_CHARGING_LEVEL, 1.0)
        .add_constant_input(RETURN_TO_CHARGE_FROM_SEARCHING, 0.1)
        .add_constant_input(FLY_TIME, 1800, is_int=True)
        .add_constant_input(RECHARGE_TIME, 3600, is_int=True)
        .add_constant_input(TRASH_PICKUP_DELAY, 5, is_int=True)
        .add_constant_input(TRASH_DROPOFF_DELAY, 5, is_int=True)
        .add_constant_input(LENGTH_OF_SIMULATION, 42000, is_int=True)
        .add_input_with_range(NUMBER_OF_COLLECTORS, 1, 10, is_int=True)
        .add_input_with_range(NUMBER_OF_CHARGERS, 1, 10, is_int=True)
        .add_input_with_range(NUMBER_OF_DRONES, 3, 27, is_int=True)
        .add_input_with_range(INIT_COLLECTORS_RANDOM, 0, 1, is_int=True)
        .add_input_with_range(INIT_CHARGERS_RANDOM, 0, 1, is_int=True)
        .add_input_with_range(PARK_SIZE, 200, 800, is_int=True)
        .add_input_with_range(TRASH_DETECTION_RADIUS, 10, 50)
        .add_input_with_range(TRASH_GENERATION_RATE, 0.003, 0.03)
        .add_input_with_range(SEARCH_PATTERN, 0, 3, is_int=True)
        .with_repeated_experiments(2, random_seeds=[53425235, 7843074239])
    )
    lhs_DOE = DOE_generator.make_latin_hypercube_doe(5000, save_to_csv=True, csv_name=EXPERIMENT_NAME)
    return lhs_DOE


if __name__ == "__main__":
    main()
```

A.3 Chapter 3 Code

many_replicates_experiment.py

```python
from doe_generator.doe_generator.design_of_experiment_generator import DesignOfExperimentGenerator
from experiment_runner.experiment_runner.parkcleanup_experiment_runner import ParkCleanupExperimentRunner
from experiment_runner.experiment_runner.string_constants import *
import time
from pathlib import Path
import pandas as pd
from copy import deepcopy
from experiment_runner.experiment_runner.path_manager import PathManager
from preferences import PATH_STRING

if __name__ == "__main__":
    OUTPUT_FOLDER = "many_replicates_experiments"
    folder_name = "long_experiments_28_april_2020"
    PathManager.BASE_PATH = Path(PATH_STRING)

    experiment_runner = ParkCleanupExperimentRunner(OUTPUT_FOLDER)
    DOE = pd.read_csv(Path.cwd() / 'data' / 'inputs' / (folder_name + '.csv'))

    inputs = []
    experiments = [11, 17, 355]
    for experiment in experiments:
        dict_ = experiment_runner._get_experiment_dict_from_pandas(DOE, experiment)
        for i in range(30):
            dict_[INDEX] = str(experiment) + " " + str(i)
            inputs.append(deepcopy(dict_))

    experiment_runner.with_csv_output_checkpointing(30)
    experiment_runner.multiprocessing_with_csv_checkpointing(inputs, 7)
```

run_difference_baseline_sims.py

```python
from paper_specific_code.analysis_paper.experiment_scripts.run_sim_parameterized import run_experiment

park_len_ref = 400
tph_ref = 40
tdr_ref = 20
num_drone_ref = 12
num_collectors_ref = 3
num_chargers_ref = 3

park_len_mod = 700
tph_mod = 70
tdr_mod = 50
num_drone_mod = 24
num_collectors_mod = 8
num_chargers_mod = 8

num_time_steps = int(3.5*24*60*60)
folder_name = "difference_baseline_experiments"
```

```
# Baseline experiment
run_experiment(num_drone_ref, num_collectors_ref, num_chargers_ref, tph_ref, tdr_ref, park_len_ref,
    num_time_steps, 0, folder_name)
# Change each experiment reference level
run_experiment(num_drone_mod, num_collectors_ref, num_chargers_ref, tph_ref, tdr_ref, park_len_ref,
    num_time_steps, 1, folder_name)
run_experiment(num_drone_ref, num_collectors_mod, num_chargers_ref, tph_ref, tdr_ref, park_len_ref,
    num_time_steps, 2, folder_name)
run_experiment(num_drone_ref, num_collectors_ref, num_chargers_ref, tph_mod, tdr_ref, park_len_ref,
    num_time_steps, 3, folder_name)
run_experiment(num_drone_ref, num_collectors_ref, num_chargers_ref, tph_ref, tdr_mod, park_len_ref,
    num_time_steps, 4, folder_name)
run_experiment(num_drone_ref, num_collectors_ref, num_chargers_ref, tph_ref, tdr_ref, park_len_mod,
    num_time_steps, 5, folder_name)
run_experiment(num_drone_ref, num_collectors_ref, num_chargers_mod, tph_ref, tdr_ref, park_len_ref,
    num_time_steps, 6, folder_name)
```

run_num_UAVS_sweep.py

```
from paper_specific_code.analysis_paper.experiment_scripts.run_sim_parameterized import run_experiment

park_len_ref = 400
tph_ref = 40
tdr_ref = 20
num_drone_ref = 12
num_collectors_ref = 3
num_chargers_ref = 3

num_time_steps = int(3.5*24*60*60)
folder_name = "NumUAVsweep"
# Baseline experiment
run_experiment(6, num_collectors_ref, num_chargers_ref, tph_ref, tdr_ref, park_len_ref, num_time_steps,
    0, folder_name)
# run_experiment(9, num_collectors_ref, num_chargers_ref, tph_ref, tdr_ref, park_len_ref, num_time_steps,
     1, folder_name)
run_experiment(12, num_collectors_ref, num_chargers_ref, tph_ref, tdr_ref, park_len_ref, num_time_steps,
    2, folder_name)
# run_experiment(15, num_collectors_ref, num_chargers_ref, tph_ref, tdr_ref, park_len_ref, num_time_steps
    , 3, folder_name)
run_experiment(18, num_collectors_ref, num_chargers_ref, tph_ref, tdr_ref, park_len_ref, num_time_steps,
    4, folder_name)
# run_experiment(21, num_collectors_ref, num_chargers_ref, tph_ref, tdr_ref, park_len_ref, num_time_steps
    , 5, folder_name)
run_experiment(24, num_collectors_ref, num_chargers_ref, tph_ref, tdr_ref, park_len_ref, num_time_steps,
    6, folder_name)
# run_experiment(27, num_collectors_ref, num_chargers_ref, tph_ref, tdr_ref, park_len_ref, num_time_steps
    , 7, folder_name)
```

```
run_experiment(30, num_collectors_ref, num_chargers_ref, tph_ref, tdr_ref, park_len_ref, num_time_steps,
    8, folder_name)
```