



NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AUTONOMOUS CUEING WITHIN HETEROGENEOUS ROBOT SWARMS

by

Britt J. Campbell

June 2019

Thesis Advisor:
Second Reader:

Duane T. Davis
Kathleen B. Giles

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			<i>Form Approved OMB No. 0704-0188</i>	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)		2. REPORT DATE June 2019	3. REPORT TYPE AND DATES COVERED Master's thesis	
4. TITLE AND SUBTITLE AUTONOMOUS CUEING WITHIN HETEROGENEOUS ROBOT SWARMS			5. FUNDING NUMBERS	
6. AUTHOR(S) Britt J. Campbell				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			8. PERFORMING ORGANIZATION REPORT NUMBER	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			10. SPONSORING / MONITORING AGENCY REPORT NUMBER	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.				
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.			12b. DISTRIBUTION CODE A	
13. ABSTRACT (maximum 200 words) When integrated into military operations, autonomous swarm technology has the potential to alleviate some of the challenges presented by demanding and highly dynamic environments and problems. Autonomous aerial swarms composed of heterogeneous robots can further increase flexibility by providing a broader range of capabilities. Heterogeneous swarm effectiveness is currently limited, however, by the system's ability to assign robots to tasks in a globally optimal manner. We propose a market-based approach to the unmanned aerial system's decision-making for allocating the most suitable robots to the available tasks. Specifically, we evaluated the performance of an auction algorithm in task allocation for an area search problem. In addition to addressing swarm heterogeneity, our implementation accounts for situations in which secondary contact investigation tasks are generated asynchronously. Experiments were conducted with the Naval Postgraduate School Advanced Robotic Systems Engineering Laboratory's aerial swarm system with various fixed-wing and rotary-wing unmanned air vehicle configurations ranging from three to ten robots. Our research and testing demonstrate that the auction algorithm is a scalable approach to task assignment in area search and suggests that our implementation can efficiently scale to swarms with arbitrary capability distributions to address highly complex problems.				
14. SUBJECT TERMS UAV, UAS, auction algorithms, market			15. NUMBER OF PAGES 213	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UU	

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

AUTONOMOUS CUEING WITHIN HETEROGENEOUS ROBOT SWARMS

Britt J. Campbell
Lieutenant, United States Navy
BS, U.S. Naval Academy, 2012

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

**NAVAL POSTGRADUATE SCHOOL
June 2019**

Approved by: Duane T. Davis
Advisor

Kathleen B. Giles
Second Reader

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

When integrated into military operations, autonomous swarm technology has the potential to alleviate some of the challenges presented by demanding and highly dynamic environments and problems. Autonomous aerial swarms composed of heterogeneous robots can further increase flexibility by providing a broader range of capabilities. Heterogeneous swarm effectiveness is currently limited, however, by the system's ability to assign robots to tasks in a globally optimal manner. We propose a market-based approach to the unmanned aerial system's decision-making for allocating the most suitable robots to the available tasks. Specifically, we evaluated the performance of an auction algorithm in task allocation for an area search problem. In addition to addressing swarm heterogeneity, our implementation accounts for situations in which secondary contact investigation tasks are generated asynchronously. Experiments were conducted with the Naval Postgraduate School Advanced Robotic Systems Engineering Laboratory's aerial swarm system with various fixed-wing and rotary-wing unmanned air vehicle configurations ranging from three to ten robots. Our research and testing demonstrate that the auction algorithm is a scalable approach to task assignment in area search and suggests that our implementation can efficiently scale to swarms with arbitrary capability distributions to address highly complex problems.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

1	Introduction	1
1.1	Background	1
1.2	The Benefits of Autonomous Swarm Robotics	3
1.3	Adding Heterogeneity to the Swarm	5
1.4	Motivation	6
1.5	Research Objectives	7
1.6	Thesis Organization	8
2	Approach	9
2.1	Research Scope	9
2.2	Assignment of Subtasks to Individual Agents	10
2.3	Related Work	12
2.4	The Auction Algorithm	16
2.5	Summary	22
3	Implementation	23
3.1	ARSENL On-UAV Software Architecture	23
3.2	Behavior Design Overview	24
3.3	Behavior Class	27
3.4	Auctioned Task Objects	29
3.5	Single Item Auction Implementation	34
3.6	Utility Function and Task Valuation	38
3.7	Intra-Swarm Messaging and Notifications	43
3.8	Summary	44
4	Experimentation	45
4.1	ARSENL Simulation Environment Experimentation	45
4.2	Expected Behavior	46
4.3	Data Collection and Analysis	51

4.4	Summary	58
5	Conclusion	61
5.1	Framework Analysis and Areas for Improvement	61
5.2	Additional Future Work.	64
5.3	Summary and Conclusions	68
	Appendix: Source Code	71
A.1	auction_searcher_test Source Code	71
A.2	auctions Source Code.	103
A.3	searcher Source Code.	129
A.4	waypoint_paths Source Code	152
A.5	independent_transits Source Code	169
	List of References	193
	Initial Distribution List	197

List of Figures

Figure 2.1	Computational complexity of centralized versus decentralized control	15
Figure 2.2	Combinatoric auction assignment example	21
Figure 3.1	ARSENL UAV onboard software architecture	24
Figure 3.2	ARSENL 10-hertz control loop	25
Figure 3.3	AuctionCue class hierarchy	27
Figure 3.4	Cued search behavior state diagram	28
Figure 3.5	Basic experimental search area depiction	31
Figure 3.6	Large experimental search area depiction	32
Figure 3.7	Waypoint sequencing for search cell coverage	33
Figure 3.8	Single item auction architecture	36
Figure 3.9	Encoding specific resource types as generic resources	39
Figure 3.10	Local utility calculation as a function of time	41
Figure 3.11	Incorporating aircraft type into utility calculations	42
Figure 4.1	UAV Airframes	46
Figure 4.2	Swarm disposition at the standby location before play initiation .	48
Figure 4.3	Dispersion of robots as they are assigned to search cell tasks . . .	49
Figure 4.4	Sample data output from a heterogeneous robot swarm	52
Figure 4.5	Task allocation across varying swarm sizes	53
Figure 4.6	Area coverage time for various swarm sizes	54
Figure 4.7	Auction resolution time boxplot	55

Figure 4.8 Average auction resolution time trend 56

Figure 4.9 Robot utilization by UAV airframe type 57

List of Acronyms and Abbreviations

ARSENL	Advanced Robotic Systems Engineering Laboratory
COTS	commercial off-the-shelf
DoD	Department of Defense
FAC	Fast Attack Craft
FIAC	Fast Inshore Attack Craft
GPS	Global Positioning System
GCS	ground control station
HADR	Humanitarian Assistance and Disaster Relief
IED	Improvised Explosive Device
MAD	Magnetic Anomaly Detection
NPS	Naval Postgraduate School
ROS	Robot Operating System
RPA	Remotely Piloted Aircraft
SITL	Software in the Loop
TCP	Transmission Control Protocol
UAS	unmanned aerial system
UAV	unmanned air vehicle
UUV	unmanned underwater vehicle
USG	United States government
USN	United States Navy

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I would like to thank my advisors, Dr. Duane Davis and Dr. Kathleen Giles, for all of the guidance they provided during my thesis project. Their thoughtful insights were helpful throughout my robotics and computer science studies. I am also grateful to the members of the ARSENL team for coordinating multiple live-fly events at Camp Roberts, CA. These exercises allowed me to see the thesis outcomes in a real-world scenario. My family and friends have been unequivocally supportive during my academic and Navy career. I would like to extend a special thanks to my wife, Jillian, whose unwavering love and encouragement has been a profound source of strength.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:

Introduction

Robot swarms are not truly autonomous unless the swarm can collectively make decisions about a globally optimal allocation of tasks to each individual robot. These decisions can often be multivariate, requiring time intensive considerations that increase in complexity as more robots with more capabilities participate in the swarm. An efficient way to automate these decisions is to have a decentralized approach that relies on distributed computation using local information to compute a globally optimal solution for the swarm. This thesis outlines our implementation of a distributed robot swarm that is robust, flexible, and scalable. The framework of our code is adaptable to a prototype the military could utilize to build a fully autonomous unmanned aerial systems (UASs) that can execute an area search with known and unknown search tasks. Although we have primarily conducted research on the coordination of unmanned air vehicles (UAVs) within a UAS for area search missions in this thesis, the implications of the results facilitate coordinated unmanned systems executing a variety of missions operating in any domain.

1.1 Background

A UAV as defined by The Department of Defense (DoD) is “a powered, aerial vehicle that does not carry a human operator, uses aerodynamic forces to provide vehicle lift, can fly autonomously or be piloted remotely, can be expendable or recoverable, and can carry a lethal or nonlethal payload and a UAS is defined as a system of UAVs, which usually consists of three or more air vehicles, a ground control station, and their associated support equipment” [1]. Congress and the U.S. military have researched UAVs since 1917 when air warfare emerged during World War I. Though UAVs did not make it into a combat role until 2002, the first practical UAVs were built as early as 1941 for a wide range of military activities. In particular, the Ryan Aeronautical AQM-34 Firebee I series exemplifies the flexibility of a UAV. Early versions of the Ryan Firebee I were remotely piloted and used primarily for gunnery target practice in 1948. Over the years, as the project evolved, the AQM-34 was adapted to an increasing breadth of missions. In 1960, the Ryan Firebee I was successfully employed throughout Vietnam and Southeast Asia as a remotely operated

reconnaissance aircraft. By 1970, Israel expanded their mission to behave as decoys and force multipliers during the Yom-Kippur war. By 2003, their mission expanded further to laying chaff corridors during the Invasion of Iraq. Technological advances in Global Positioning System (GPS) navigation systems, and the availability of communications systems as complex as broadband satellite links have enabled similar proliferation of a variety of other UAS [2]. In practice, many of these UAVs such as the Ryan Firebee are more specifically categorized as Remotely Piloted Aircrafts (RPAs). These aircraft are unmanned in the sense that there is no pilot on board but the aircraft still require an aircrew at a remote piloting station.

Developments in UAS technology naturally preceded research interests in creating a fully autonomous UAV swarm of multiple robots. Autonomous swarms alleviate the burden of manning the UAS remotely but they do not come without their own challenges and specifications. The notion of an intelligent swarm is analogous to the swarming behavior of social insects. There is no centralized coordinator behind their behavior, yet they organize themselves *intelligently* to accomplish tasks that broadly benefit their colony. Intelligent behavior, as defined by Gerardo Beni, is one that produces an ordered, unpredictable, and improbable outcome [3]. The behavior must result in a pattern that reflects progress toward a common goal. It should be unpredictable because if it follows a methodical, repeatable process, then the swarm is not truly intelligent and will not be adaptable to new scenarios. Lastly, a satisfactory outcome must be improbable or else it is just as likely to occur from an unintelligent swarm behaving randomly.

In [4], Erol Sahin defines swarm robotics as “the study of how a large number of relatively simple physically embodied agents can be designed such that a desired collective behavior emerges from the local interactions among agents and between the agents and the environment.” Ideally, the collective behavior that is produced from robot coordination leverages the capabilities of individual robots to obtain results that would be difficult or impossible for a single robot to achieve on its own. He further explains that in order to successfully implement a robotic swarm that fully leverages the advantage of coordinated robotics, the swarm should be robust, flexible, and scalable. Robustness means that the swarm should be resistant to failure by individual participant robots. Redundancy is one method for increasing robustness but a better approach that we discuss throughout this thesis is decentralized control over the execution of tasks. Remarkably, robot swarms can produce highly complex

outcomes even if the individual robots behave in a simple manner. They are robust because autonomous robots with distributed behavior algorithms will continue to operate regardless of any single failure elsewhere in the swarm. It is important to note that swarms should be robust to loss in the event that it happens but building robots with the computing power to be autonomous that are intended to be expendable is an engineering challenge that is outside the scope of this thesis. Flexibility refers to the range of tasks to which a robot swarm can be adapted. Particularly in the military, objectives are dynamic and rapidly evolving. Building autonomy into the swarm requires that it is flexible enough to operate in uncertain environments. Lastly, the swarm needs to be scalable. The UAS should be able to perform similarly in large groups as it would in smaller groups. Arguably, effective scaling of a robotic swarm necessitates a level of autonomy, especially if the swarm is built with decentralized control.

The characterization of a swarm as robust and flexible aligns with a recurring theme in modern Combatant Commanders' interest in leveraging swarm technology for its persistent resilience, its versatility and its ability to create a robust interoperable foundation for operations. The DoD recognizes that UAS technology can make our fighting force more lethal and optimize routine military tasks. Our implementation of the auction algorithm sought to meet the standards of an intelligent robot swarm set forth by Sahin and Beni, while demonstrating a feasible solution to today's military research initiatives in robotics.

1.2 The Benefits of Autonomous Swarm Robotics

An autonomous robot is one that has awareness of its position in the world and is able to interact with its environment independently of human interaction [4]. Bringing autonomy to the individual robots enables the scalability and flexibility of the robot swarm. Scaling from one vehicle to multiple vehicles in the swarm can be overwhelming for the operator unless the robots have sufficient autonomy. If autonomy can be achieved, many tasks that are appropriate for one vehicle manned by one operator are expedited by multiple autonomous coordinated vehicles.

Area coverage is a common challenge that is suitable for swarm robotics and it is the focus of this thesis. Consider a relatively large region that needs to be surveyed or monitored closely. In this simple problem, the challenge is to organize the swarm to systematically search a

known area while minimizing the amount of time (or some other valuable resource) to complete the survey. Our thesis takes this question further by adding a secondary challenge. If the area is only partially known, the swarm must also have a mechanism for handling secondary situations that it discovers as a result of its search and it must be able to reorganize itself throughout the execution of the search to accommodate these discoveries. If the individual robots are autonomous, they will all know where they are in the world and will be able to interact with the search region and each other. Creating the algorithm by which they will organize among themselves to execute the search methodically is our goal.

Autonomy facilitates scalability. A limiting factor in area coverage is the endurance of the individual robot. One robot will have to refuel or recharge more frequently than a group of coordinated robots. The larger group allows the swarm to more efficiently complete larger search areas. Distributed algorithms scale well because each individual robot autonomously organizes itself within the swarm. Another aspect of scalability is the idea of scaling in time discussed by Erol Sahin in [4]. Scaling in time is the idea that an autonomous swarm can seamlessly continue its task at hand if more robots are added to the swarm during execution. Continuing the area search example, consider an individual robot encountering a dangerous combat situation that requires additional resources to successfully engage the threat. A single operator could certainly get overburdened by the additional robots and required attention to the threat. If swarm frameworks include a way to incorporate the new robots in the organization of their task execution, they can scale as needed to adapt to evolving, complex problems. Scaling in time is not required for a robot swarm to meet the standard of being scalable but it is a feasible capability of swarm architecture that is not possible to implement in RPAs.

By virtue of being able to complete larger tasks more efficiently, robot swarms are inherently more flexible than single-vehicle systems. They can also be adapted to a wider range of tasks if they are autonomous. RPAs require an operator who is familiar with the environment and trained on the robots' performance when operating in different missions. Robots that have an awareness of where they are and that can make decisions autonomously about how to interact with their environment can make these adjustments automatically. In some cases, software has to be updated to support the new tasks, but it can be simpler than teaching an operator. We have developed a modular architecture. We describe how tasks are defined for the framework in Chapter 3 in a simple manner that promotes flexibility within the swarm.

1.3 Adding Heterogeneity to the Swarm

Conventional robot swarms are homogeneous; all the robots within the swarm have similar capabilities and limitations, and they can all perform tasks more or less interchangeably, producing similar outcomes. In [4], Sahin argues that the parallel between robotic swarm behavior and biological swarm behavior falls apart if a large amount of heterogeneity is allowed. In [5], Lynne E. Parker identifies an environment where certain tasks can only be accomplished by robots with certain capabilities. In such an environment, the heterogeneity is the result of multiple homogeneous sub-swarms of robots that will only execute specific tasks that are appropriate for their capabilities. For example, consider a swarm of UAVs and unmanned underwater vehicles (UUVs). The UAVs are incapable of carrying out tasks that can be assigned to UUVs and vice-versa. On the other hand, heterogeneous robots can be employed in an environment where there is overlap in the robots' capability to accomplish tasks.

In this work we used fixed-wing UAVs and rotary wing UAVs; similar robots with similar capabilities. Tucker Balch describes a metric for the level of heterogeneity in a swarm in [6]. This thesis is not concerned with a precise measurement of heterogeneity, and it was sufficient for our research to pick robots that are closely related. Each platform is capable of accomplishing any of the available tasks, but one aircraft is often better suited to a task than the other based on the parameters we established.

Trying to build a flexible homogeneous swarm of robots presents two problems that are enumerated by Parker in [7]. The first problem has to do with the engineering considerations of the robots. It may be pragmatically prohibitive to build a UAV with all of the necessary features and capabilities to optimally execute every task in the environment. It may be more economically viable to build multiple specialized types of robots. Even if it is possible to build homogeneous robots that are sufficiently capable of handling every task, heterogeneity can still emerge if any of the robots are partial toward certain types of tasks as a result of the underlying assignment algorithm. Alternatively, if it is not a necessity, heterogeneity might be a nice feature that adds predictability, increased robustness, or greater overall control to the swarm. The second problem is that strict homogeneity presents a scaling problem. It is difficult to create two robots that are exactly identical with the same sensors, performance characteristics, and calibrations, and the problem multiplies in larger swarms. Material maintenance is exacerbated over time as the robots experience physical degradation. Rather

than accounting for the diversity from these inevitable imperfections after they develop, heterogeneity should be a design consideration of the swarm at the outset.

The added challenge of a heterogeneous swarm is that there must be a distributed task assignment algorithm to help the swarm decide how robots should be allocated to tasks in an efficient manner. Our goal, therefore, was to demonstrate how using the auction algorithm can alleviate some of the limitations in the flexibility of a homogeneous swarm while evaluating a solution to task assignment in a heterogeneous implementation.

1.4 Motivation

Heterogeneous search swarms are more dynamic than homogeneous search swarms if they are provided ability to reorganize themselves to maximize the pairing of robots to tasks based on the attributes of the robots. Military operations are inherently dynamic. Whether ships are organized in a multi-platform carrier strike group or Marines are organized into a battalion landing team, the environment can change rapidly and the support systems in place must be able to react to provide the appropriate support that the units require. The only organic option a modern Arleigh Burke class destroyer has to help search for submarines are two helicopters, such as the MH-60R Seahawk, that are normally cued by operators within the ship or P-8 Poseidon fixed-wing aircraft. Turkish Navy Lieutenant Omur Ozdemir was able to conclusively demonstrate that a ship's capability to find submarines was greatly increased by integrating airpower from helicopters [8]. Helicopters are an effective force multiplier for an independent surface ship. A similar bolstering effect can be duplicated with a distributed UAS swarm.

Robot swarm search has humanitarian applications as well. One of the challenges of providing adequate disaster relief, for instance, is identifying areas that are the most affected by the natural disaster. A preliminary step in all search and rescue efforts is surveying the area to map out the extent of the damage. In the midst of a crisis, persistent high resolution aerial imagery provides information to first responders about where aid is needed most and how best to provide the aid. After the disaster has occurred, aerial imagery can provide information about the land's resources that can assist with the recovery such as water sources and construction materials. Aerial imagery can also identify geographical hazards that continue to pose a threat after the disaster has subsided. [9]. In many situations,

critical infrastructure is severely damaged or destroyed, precluding conventional vehicles from conducting these invaluable surveys and searches. UAV swarms can circumvent the infrastructure altogether, providing consistent updates to the situational awareness that responders require to provide aid.

1.5 Research Objectives

In a fully autonomous robot swarm, at least two types of robots will be required for the different tasks the robot swarm may have to accomplish. This thesis will focus primarily on optimally searching an area. In reality, thoroughly searching an area is multifaceted. Individual robots within the swarms that are searching will be intended for use in unfamiliar and potentially austere environments. As more information is gleaned about the environment, the swarm needs to adapt by allocating robots to tasks in a manner that is globally optimal given the updated information. In particular, some robots may be more suitable for certain tasks and those robots should be allocated to tasks that they are more functionally capable of accomplishing.

Autonomous coordination within a heterogeneous swarm to complete a complex area search was demonstrated in [10]. This implementation utilized the auction algorithm, a method for solving the assignment problem developed by Dimitri Bertsekas [11]. More specifically, Hopchak utilized the auction algorithm to determine optimum search cell assignments that accounted for individual vehicle speed and endurance.

In an effort to expand the applications of Hopchak’s research, we show that the algorithm is appropriate for a more generalized assignment problem. In particular, we explore a search scenario in which the swarm conducts an area search that is interrupted by “detection” events that require investigation. That is, as the search progresses, robots that detect potential contacts of interest independently generate cuing events that initiate a new auction including both the remaining search cell tasks and the contact investigation tasks. Further, individual swarm members will have different capabilities that make them more suitable to either the search or investigation the task. We demonstrate that the auction algorithm is an effective way to obtain near-optimal assignments of members of this heterogeneous robot swarm to a dynamic set of heterogeneous tasks. Extension of this approach beyond the cued search implementation of this thesis will prove adaptable to a broad array of complex tasks.

1.6 Thesis Organization

This thesis is organized into five chapters and an appendix with the source code. Chapter 1 provides the background information to the robot swarms and discusses works related to this thesis. Chapter 2 describes the research goals and problem statement of the thesis as well as the practical approach we have taken to solving the problem. Chapter 3 enumerates the specifics of our implementation and how we have designed our experiment. Chapter 4 presents and analyzes the results of our experiment. Chapter 5 analyzes the implications of the results and describes areas of improvement on our results as well as the implications to future work based on our findings.

CHAPTER 2:

Approach

2.1 Research Scope

The main goal of our work was to leverage the efficiency and distribution of an auction algorithm to determine a near-optimal division of labor between the members of a heterogeneous aerial robot swarm completing a complex dynamic task (e.g., an area search that will yield contacts of interest requiring further investigation). Following task allocation, swarm UAVs complete their assigned tasks while also taking into consideration how the environment changes over time. The overall goal was to achieve a globally near-optimal assignment of heterogeneous robots to heterogeneous tasks. Requisite to our primary goal, we demonstrate that our approach finds a near-optimal solution in every case using a market based approach to the assignment problem.

To achieve this objective, we use a variation of Bertsekas’ auction algorithm as described in Section 2.4. In particular, we extend previous work in which Bertsekas’ algorithm was used to obtain near-optimal search cell assignments [10]; however, we modify the bidding and auction processes to accommodate the ability of the swarm to leverage task-specific capabilities of individual UAVs. Our implementation autonomously controls a swarm of UAVs with two vehicle-type-specific capabilities (i.e., optimization for search or investigation) and speeds. We assert that our implementation can efficiently scale to swarms with arbitrary capability distributions (i.e., more than two).

In addition to addressing swarm heterogeneity, our implementation successfully accounts for a dynamic system in which secondary tasks are generated asynchronously. In other words, once a swarm UAV identifies a new task, that task will be incorporated into the task allocation algorithm. Furthermore, current task execution can be truncated in cases where UAVs determine that they are more suited to an emergent task than their current one.

Finally, our implementation is used to evaluate ways to formulate task valuation within the bidding framework to identify those most likely to yield near-optimal solutions.

2.2 Assignment of Subtasks to Individual Agents

The problem of assigning a number of agents (e.g., robots in a swarm) to a set of discrete tasks based on the most optimal allocation of robots to tasks is known as the assignment problem. In 1979, Dimitri Bertsekas presented the auction algorithm as an effective method for solving this problem [11]. Much like a conventional auction, Bertsekas' algorithm has each agent bid for tasks based on local utility values that the agent associates with completion of specific tasks. We incorporate a version of the auction algorithm into the Naval Postgraduate School (NPS) Advanced Robotic Systems Engineering Laboratory (ARSENL) UAVs swarm and evaluate its effectiveness in controlling a heterogeneous swarm through an aerial search of a large area to include investigation of identified contacts of interest.

2.2.1 The Basic Assignment Problem

The goal of any linear programming problem is to achieve a maximum (or minimum) outcome. The assignment problem often specified as a linear programming problem that optimally allocates agents to tasks where optimality is measured by the cost of accomplishing a task and the benefit the task's completion has to the swarm.

First, consider a symmetric assignment problem in which the number of agents and available tasks are equal. That is, n agents are to be assigned to n tasks. There is a set of utility values a_{ij} for matching each agent i to each task j , and the goal is to assign agents to tasks so that the global utility is maximized. Associations between a task and an agent are denoted with an indicator variable x_{ij} where $x_{ij} = 1$ if the agent is assigned to a task or $x_{ij} = 0$ if the agent is not assigned to the task. The basic assignment problem is frequently specified as a linear program maximizing the total benefit of the assignments as follows [12], [13]:

$$\max \sum (a_{ij}x_{ij}) \quad (2.1)$$

Assignments are made subject to the constraints that no agent can be assigned to more than one task and no two agents can be assigned to the same task:

$$\sum (x_{ij}) = 1, \forall i = 1, \dots, n \quad (2.2)$$

$$\sum (x_{ij}) = 1, \forall j = 1, \dots, n \quad (2.3)$$

A final constraint is that agents cannot commit to partially completing tasks. They either take the whole task $x_{ij} = 1$ or not at all $x_{ij} = 0$.

$$x_{ij} \in \{0, 1\} \quad (2.4)$$

A brute force method for solving this problem might exhaustively assign agents to tasks until all possible combinations are explored. Solutions are identified as those combinations that are maximal (there may be more than one).

Algorithm 1 Calculate $\max \sum(a_{ij}x_{ij})$

```

a =  $n \times n$  cost_matrix
for all row, r, in a do
     $v_{min} = \min(v \text{ in } r)$ 
    for all utility value, v, in r do
         $v = v - v_{min}$ 
    end for
end for
for all column, c, in a do
     $v_{min} = \min(v \text{ in } c)$ 
    for all utility value, v in c do
         $v = v - v_{min}$ 
    end for
end for
while True do
    num_lines = cover all 0's with as few lines as possible
    if num_lines == n then
        return a
    else
         $v_{min}$  = smallest uncovered utility cost
        subtract  $v_{min}$  from all values in each uncovered row
        add  $v_{min}$  to all values in each covered column
    end if
end while

```

A more efficient approach to solving the basic assignment problem is provided by the Hungarian Method (Algorithm 1). This algorithm relies on the theorem that “If a number is added to or subtracted from all of the entries of any one row or column of a cost matrix, then an optimal assignment for the resulting cost matrix is also an optimal assignment for

the original cost matrix.” [14], [15]. By systematically subtracting the value of the lowest cost from all other rows in a column, a globally optimal assignment can be discovered.

2.2.2 The Generalized Assignment Problem

In Section 2.2.1, the assignment problem is defined symmetrically. That is, the number of agents equals the number of tasks, so each agent is assigned to exactly one task. The generalized assignment problem is a useful extension of the basic assignment problem that models a scenario where n agents are to be assigned to m tasks where n does not equal m . In this scenario, each agent can be assigned to more than one task. As with the basic assignment problem, the generalized assignment problem seeks to maximize the total system utility as defined by the sum of the individual assignments [16], [17]:

$$\max \sum (a_{ij}x_{ij}) \quad (2.5)$$

Assignments are made subject to similar constraints as those of the basic assignment problem except that each agent is assigned to a (possibly empty) subset of the available tasks:

$$\sum (x_{ij}) = 1, \forall i = 1, \dots, n \quad (2.6)$$

$$\sum (x_{ij}) \geq 0, \forall j = 1, \dots, m \quad (2.7)$$

Agents are still prohibited from being assigned to partial tasks.

$$x_{ij} \in \{0, 1\} \quad (2.8)$$

2.3 Related Work

2.3.1 Centralized Subtask Assignment

When a team of robots is searching an area, it should conduct the search in an organized, efficient manner. One approach to controlling a robot swarm is to think of the swarm as a single unit performing a large task that is divided into subtasks by a controlling station [18]. If the controlling station divides the area into a number of sub-areas (i.e., search cells) equal to the size of the swarm, the assignment of individual swarm members to search cells can

be specified using the basic assignment problem and the central controller can utilize the Hungarian Algorithm to determine the optimal individual assignments. If the area division results in a number of search cells not equal to the swarm size, the problem is expressible as the general assignment problem and requires a more robust solution to optimize the assignments.

Our research is not formally defined in terms of the generalized assignment problem because not all the tasks are known to the robots at the beginning of the search. As discussed in Section 1.5 some of the tasks are generated asynchronously and the swarm must reorganize itself to accomplish them. We do, however, use the generalized assignment problem as a foundation from which we can extrapolate to define the more complex cued search problem and approaches for solving it.

With a centralized approach to sub-task assignment, the controller maintains a global view that facilitates the formulation of global plans. There is a level of organization and predictability provided by centralizing all of the robots' control mechanisms, but achieving this level of control requires a number of prerequisites. First, in order to make decisions about what each individual robot should do, the controller needs to maintain the status of all the robots at all times. In order to do this, the controller must maintain constant supervision of all of the swarm robots through periodic updates and communication or direct observation. Second, the controller requires a robust algorithm to determine the globally optimal allocation of tasks to robots that scales gracefully as swarm size increases (Hungarian Algorithm complexity is $O(n^3)$ for a swarm size of n [15]). Though solutions to these challenges exist, they expose the swarm to computational complexities, vulnerabilities, and inflexibilities. Furthermore, they are counter to the notion of distributed decision-making within the swarm.

One glaring vulnerability of a centralized approach is that the controller is a potential single point of failure. To ensure that the communication between the controller and the swarm is never interrupted, redundancy must be incorporated to provide robustness. These redundancies introduce inefficiencies and possible race conditions that can affect performance: either a secondary controller has to maintain all of the same information as the primary controller or some mechanism must exist for the secondary controller to recover information about the swarm and the environment if the primary controller fails. As swarm

size increases, the additional communications required to support redundancy can lead to communication link saturation.

Another challenge with taking a centralized approach is associated with task assignment itself. Centralized algorithms for solving the generalized assignment problem are in the class NP-hard, meaning that they become intractable as swarm size increases [19]. Even the polynomial complexity of basic assignment problem solutions (e.g., the Hungarian Algorithm has a computational complexity of $O(n^3)$ [15]) and generalized assignment problem approximations can impose unacceptable computation requirements on a centralized controller. Swarm heterogeneity only exacerbates these problems as the different capabilities are considered and the problem becomes more complicated.

2.3.2 Decentralized Subtask Assignment

An alternative to the centralized approach described in Section 2.3.1 is to allow each robot to maintain its own utilities for available tasks for use in the local determination of task assignments. A decentralized system eliminates the single point of failure associated with centralized approaches, leverages distribution to mitigate algorithmic complexity, and accounts for individual robot failures more gracefully. In a decentralized system, each agent manages its own tasks and makes decisions about assignments locally. Agents exchange information as necessary to maintain distributed situational awareness until all the robots have agreed upon a satisfactory assignment of tasks.

Market-based algorithms provide one approach to distributed task assignment that has shown promise. This type of system envisions the multi-agent system as an analog to individuals operating in a free market environment. Agents exchange “pricing” information about potential task assignments, and then each agent makes a decision based on a selfish optimization of its own utility. This approach attempts to obtain a near-optimal global solution by locally optimizing each individual agent’s utility.

A significant benefit of market-based approaches is that the worst-case computational complexity of these algorithms is $O(n^2 \log n)$, and they can achieve linear run-time performance in many cases [20]. Implementations of centralized approaches, on the other hand, have worst-case complexity of between $O(n^3)$ and $O(n!)$ [15], [19]. As indicated in Figure 2.1, market-based algorithms are more scalable than centralized approaches and can be expected

to perform better as swarm size increases.

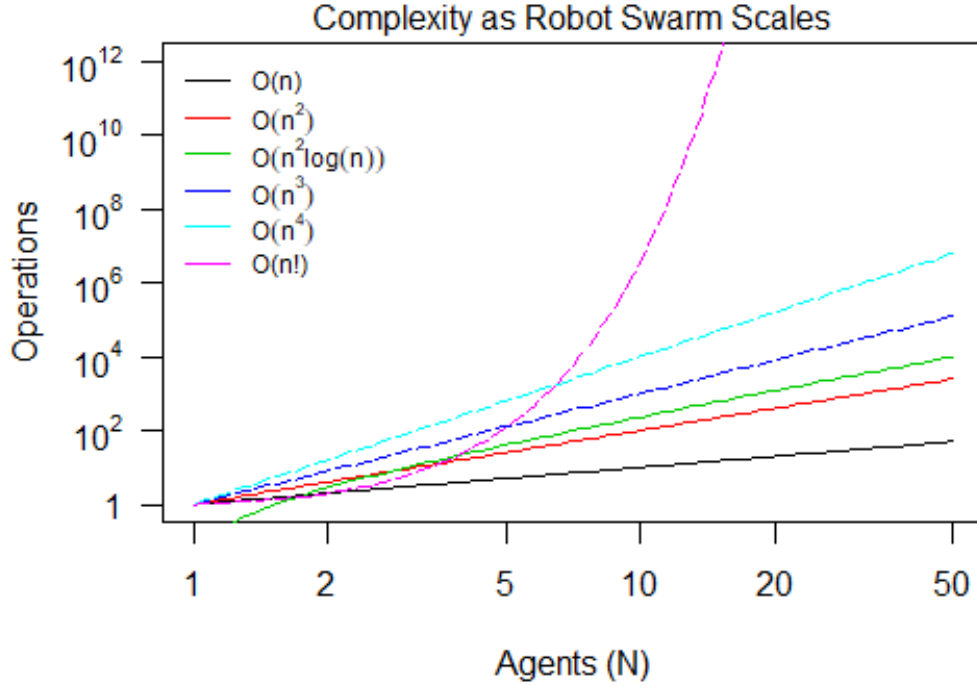


Figure 2.1. Centralized controller approaches (dashed lines) to the basic and generalized assignment problem have computational complexity of $O(n^3)$ and $O(n!)$ respectively whereas a market-based approach (solid lines) can achieve solutions in $O(n^2 \log n)$ time or better. The scalability of a swarm system is compromised using a central controller.

Our implementation utilizes this distributed, market-based approach to enable an aerial swarm to efficiently search a large area and investigate contacts that are generated over the course of the search. This approach demonstrates robustness in the event of individual agent failures and eliminates single points of failure. Additionally, our approach minimizes communications and computational bottlenecks associated with centralized approaches.

2.3.3 Inter-Vehicle Coordination

A swarm system can be organized around either tight or loose inter-vehicle coordination [21]. Tight coordination among robots requires that they actively coordinate throughout the execution of a group task. In tightly coordinated swarms, tasks must be accomplished in a specific order or multiple tasks must occur within a specific time interval or simultaneously.

A distributed allocation of tasks based on specific constraints and predefined execution plans can accomplish tightly coordinated robotic swarm assignments as demonstrated in [22] and [23]. This implementation does not use a market-based distributed system, because the tightly coordinated tasks require a level of oversight that distributed algorithms do not provide to ensure the robots complete the tasks in the established order every time.

Loose coordination, on the other hand, only requires that the robots coordinate during the allocation of the tasks. Execution of assigned tasks are completed with minimal coordination between robots. In other words, loosely coordinated tasks are largely independent of each other, and the only coordination between the tasks occurs in the decomposition of the global task into smaller tasks and the assignment of those tasks to individual swarm agents. Once the tasks are assigned, each individual agent takes full responsibility for executing the task from start to finish. Loose coordination is appropriate for the cued search problem addressed in this research since the global task can be divided into search cell and contact investigation tasks that are easily executed by a single robot. Furthermore, the discrete nature of these tasks makes a market-based approach to their assignment feasible.

2.4 The Auction Algorithm

Dimitri Bertsekas proposed such a market-based distributed method for solving the basic (and with modification, the generalized) assignment problem that he aptly names the Auction Algorithm. The basic single-item auction is conducted as a series of rounds and terminates when the linear program describing the allocation problem is satisfied. A single auction round as described in [11], [13], [24], [25] is depicted as Algorithm 2. The algorithm closely imitates a live auction except that agents exchange bids directly rather than relying on an auctioneer, thus removing the requirement for a central controller. Each agent bids to maximize its own private utility at minimum cost. As agents make their bids, available task prices rise and the agents' net utility values change accordingly. Upon conclusion of the algorithm, each agent will be allocated the best task given its local task valuations and the current pricing. The resulting assignment is near optimal and the algorithm's complexity is pseudo-polynomial [11], [13].

Algorithm 2 Basic Algorithm for Conducting a Single-Item Auction

```
S ← [(resource0, cost0, assignmentm), ... , (resourceIDm, costm, assignmentm)]  
B ← [bidder0, ... , biddern]  
assignments ← []  
for j = 0 to length(S) do  
  bids ← []  
  for i = 0 to length(B) do  
    bid ← calculate_bid(bidderi, Sj)  
    bids.append(bidderi, bid)  
  end for  
  highBid, winner ← winner_determination(bids)  
  assignments.append((resourcej, highBid, winner))  
end for  
return assignments
```

2.4.1 Local Item Valuation

As discussed, in Section 2.2, the basic assignment problem attempts to assign n agents to n tasks while requiring that each agent i be assigned to exactly one task j . The generalized assignment problem, on the other hand, attempts to assign n agents to m tasks while requiring that each task be assigned to exactly one agent. The cued search problem addressed by this thesis more closely aligns with the generalized assignment problem since the number of UAVs being tasked does not equal the number of available tasks (i.e., search cells and contact investigations). We nevertheless compute agent-specific utility values for each task individually rather than for subsets of available tasks.

At the beginning of an auction round, each agent, i , computes a private utility value a_{ij} for each task, j . Each task has an associated price, p_j , based on the most recent bid that is factored into the private utility value. Bertsekas [26] defines the condition “happy” as a situation where an agent, i , is assigned the task, j , that maximizes its local utility value:

$$assignment_i = \arg \max_j \{a_{ij} - p_j\} \mid j = 1, \dots, m \quad (2.9)$$

The overall valuation of an assignment is computed as the agent’s local utility value for the task a_{ij} minus the current price of the task, p_j . The agent is “happy” when it is assigned a task that provides the greatest possible valuation. In each round agents place bids on the

tasks that provide the greatest valuation. The auction terminates when the swarm reaches a state where all agents are “happy” with their assignments. Upon termination, the current assignments approximate an optimal solution to the assignment problem.

2.4.2 Bidding Process

Bertsekas defines an exchange-based approach whereby individual economic agents bid under the item valuation constraints discussed in the previous Section, seeking an assignment with which the agent would be “happy” in [11]. The bidding process commences with each agent being assigned to an arbitrary task at 0-cost. In turn each agent identifies the task offering the highest valuation according to Equation 2.9 and trades for the task with the agent that currently holds the task (or simply takes it if it is unassigned). Next the agent updates the price for the task to reflect the level at which that agent would be indifferent between the maximal task and the second maximal task:

$$p_j = p_j + \gamma_i, \quad (2.10)$$

where

$$\gamma_i = v_i - w_i, \quad (2.11)$$

where v_i is the maximal task value,

$$v_i = \max\{a_{ij} - p_j \mid j = 1, \dots, m\} \quad (2.12)$$

and w_i is the second maximal task value,

$$w_i = \max\{a_{ik} - p_j \mid k = 1, \dots, m \wedge k \neq j\} \quad (2.13)$$

Consequently, agents that have unfavorable secondary options will set the price very high, and agents who have favorable secondary options will set the price lower. Thus, in the distributed market, agents who do not have many options will bid more aggressively for

their best choice than those with multiple options.

Unfortunately, [11] identifies a situation in which bidding as described above will not lead to convergence on a solution. In order for the algorithm to terminate, the prices must monotonically increase to prevent agents from indefinitely exchanging the same tasks. Bertsekas' algorithm ameliorates the potential deadlock by requiring that a bid increase the price of a task by some positive scalar value ϵ to enforce monotonic price increase. The bidding equation defined in Equation 2.11 is modified slightly to include the mandatory price increase as:

$$\gamma_i = v_i - w_i + \epsilon \quad (2.14)$$

If each individual economic agent bids in this fashion, the algorithm will find an optimal solution approximation to the linear optimization problem when every agent is within ϵ of being "happy" according to the following Equation:

$$\max \sum (a_{ij} - p_j \geq \max\{a_{ik} - p_k\} - \epsilon) \mid i, j = 1, \dots, m \quad (2.15)$$

Dimitri Bertsekas went on to prove that the solution would be found within $n\epsilon$ of being optimal in [13] and discussed a parallel implementation of the algorithm in [26]. The algorithm has pseudo-polynomial complexity; the worst case run-time is dependent on ϵ , the number of tasks and the maximum absolute utility for an agent. Bertsekas has shown that if these variables are carefully selected, the algorithm converges in nearly linear time.

2.4.3 Extended Single Item Auctions and Dynamic Cueing

In typical execution, agents continue to bid for tasks until there are no more tasks remaining. We extend this conventional assignment approach to address problems wherein the tasks dynamically change. Consider a set of tasks that changes as tasks are executed. In these situations, it may be advantageous for agents to change their task allocation given new information about the environment. Specifically, the area search scenario we are considering must account for the investigation of contacts that are detected over the course of the search. Bertsekas' implementation lends itself well to these detection events through the creation

of new investigation tasks for assignment via subsequent auction within the ARSENL UAV swarm system.

We use Bertsekas' auction algorithm to dynamically assign robots to tasks in a near optimal manner by applying a simple extension to the single item auction. In a system with fewer agents than tasks, some tasks may remain unassigned after the initial auction. Upon completion of an assigned task by a robot, a new auction is initiated for tasks that are still available. This process continues until all tasks have been assigned and their execution has begun. Toward the end of the search, it is likely that there will be more agents than remaining tasks. In that case, the system inserts "null" tasks with no value into the auction so that all participating agents are assigned efficiently (i.e., null tasks will be assigned to robots not well positioned or suited for the available real tasks).

Our implementation for this simple extended auction is designed to facilitate expansion to account for investigation tasks that are asynchronously added to the system. As robots generate detection events, they add investigation tasks that robots bid on in the same manner as search cell assignments. In addition, a detection event triggers a new auction so that it can be accounted for immediately. The straightforward object-oriented framework implementing this functionality is described in Section 3.3 and Section 3.4.

2.4.4 Combinatoric Auction

One way to increase the optimality of the result of the auction is to allow agents to bid for multiple tasks in one auction round in what is called a combinatoric auction. The added advantage to a swarm system that allows agents to bid for a collection of tasks at one time is that individual robots can make decisions that systematically group tasks together to allow seamless progress between tasks. Search cells, for example, can be assigned to facilitate transition from one cell to the next as depicted in Figure 2.2.

The drawback is in the efficiency of finding a solution, because the number of potential assignments that each agent must consider is factorial [27]. For example, one agent has to compare the utilities of all of the single-item allocations with the best dual-item allocations to the best three item allocations, up to the number of tasks minus the number of agents. A single agent considering the best allocation of three ordered tasks from ten choices will have to consider 120 options before moving on to a four task grouping in accordance with

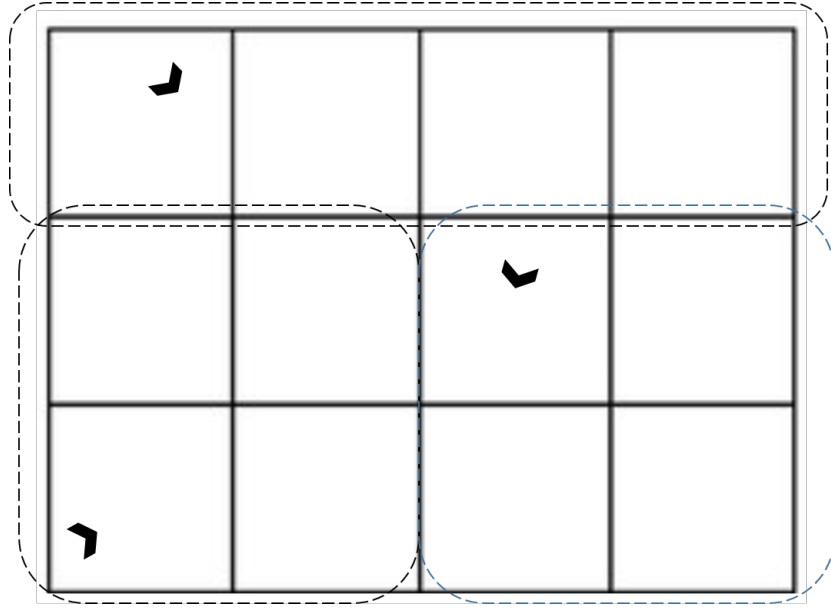


Figure 2.2. In this example of a combinatoric assignment, UAVs are assigned the four search tasks that are closest to them that do not intersect with other robots' assignments; however, finding this assignment in an automated manner has factorial complexity.

the binomial coefficient equation:

$${}_nC_k = \frac{n!}{(n-k)!k!} \quad (2.16)$$

For each numbered task grouping, the agent will have to perform this operation, continually maintaining data about the relative utility of each permutation. Furthermore, robots have to deconflict overlap in task allocations so the bidding mechanism must ensure that no subset of any agent's allocation is a subset of any other agent's allocation. Thus, unconstrained, as the swarm grows, the problem of assigning robots to tasks becomes increasingly complex with a factorial growth resulting in a combinatorial explosion that limits the scalability of a combinatoric approach; this is the crux of the NP-hardness of the generalized assignment problem.

2.4.5 The Utility Function

The underlying foundation of implementing a sensible swarm behavior using an auction algorithm is a utility function that accurately reflects the relative values of the tasks to the agent. How the agents bid is dependent on the locally calculated values of the tasks. Bids that inaccurately reflect the relative values of the available tasks will result in assignments that do not exhibit the intended behavior. As part of our research, we, through theory and experimentation, identify a utility function that fosters desirable global swarm behavior in the completion of the area search with cued investigations. The specifics of our utility function are described in Section 3.6.

2.5 Summary

This Chapter provided an overview of our use of Bertsekas' auction algorithm to determine a near-optimal division of labor between the members of a UAV swarm based on the nature of the task and distributed assessment of each robot's suitability for each of the subtasks. This approach uses a vehicle-specific local utility valuation for each task to inform individual bidding strategies. As described, our single-item auction implementation is conducive to graceful incorporation of environmental changes and allows the swarm reallocate tasks as required. Chapter 3 details the implementation of the auction algorithm on the ARSENL UAV swarm.

CHAPTER 3: Implementation

This chapter provides an overview of the ARSENL on-board software architecture and our single-item auction as implemented by the source code in the Appendix. Following an overview of the on-board software architecture, the auction algorithm implementation is described. This description first provides a broad overview of the algorithm and its execution flow. Subsequent sections describe how tasks are defined and manipulated for use in the auction, how the auction is conducted to include bid exchange and information retention, how local task utility values are calculated, and how robots simultaneously handle auction and task execution.

3.1 ARSENL On-UAV Software Architecture

All ARSENL on-UAV software is written in Python 2.7 and utilizes the Robot Operating System (ROS) to facilitate inter-process communication. Functionality is divided among a set of ROS nodes that each serve a specific purpose, as depicted in Figure 3.1. Coordinated swarming behavior, including the cued search described in this thesis, is incorporated into the *Swarm Control* node using a plugin architecture. *Network* and *Autopilot* nodes respectively support inter-vehicle and UAV-ground control station (GCS) communication and interaction with the Pixhawk autopilot. Additional nodes are implemented to provide for swarm-wide situational awareness and vehicle safety [28].

Commands are derived by the *Swarm Controller* node and relayed to the *Autopilot* node at a rate of ten hertz according to Figure 3.2. During each iteration of the control loop, the *Swarm Control* node conducts status and safety checks and then invokes the active behavior’s *computeCommand* method. The computed command is then checked by the *Safety* and *Collision Avoid* nodes prior to relay to the *Autopilot* node. An active behavior is immediately deactivated if the *Swarm Controller* node or the active behavior itself detects a problem. When the *Safety* or *Collision Avoid* node identifies a problem, the behavior-generated commands will be overridden until the situation is resolved. In some cases, the *Safety* node may disable swarming (i.e., behavior execution) and direct the vehicle to a predetermined safe location.

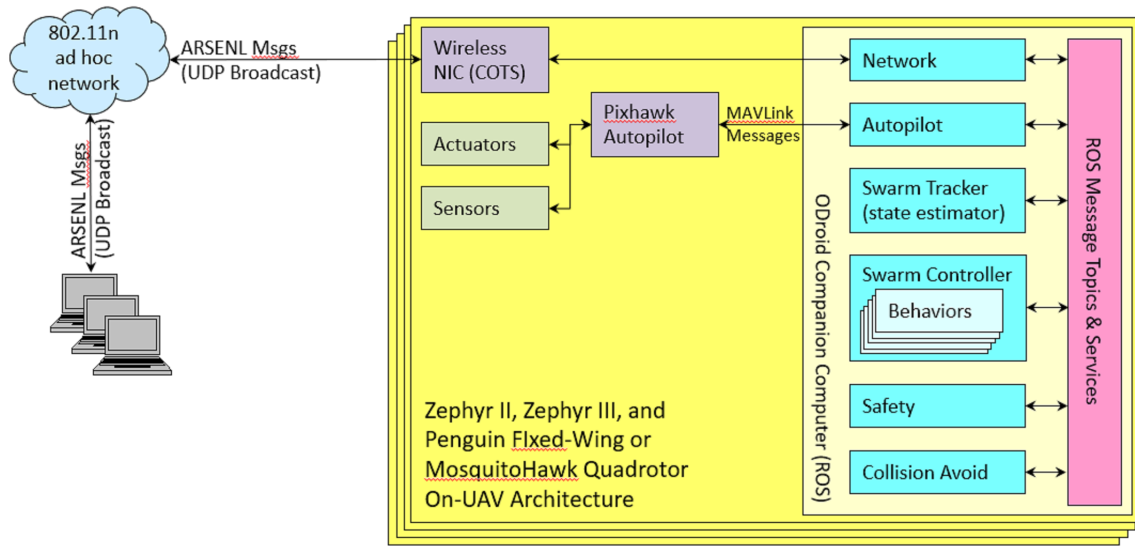


Figure 3.1. ARSENL onboard functionality is implemented as a set of ROS nodes. Behaviors are loaded into the *Swarm Controller* node as plugins.

3.2 Behavior Design Overview

In this section, we specifically describe the implementation of the cued search behavior within the ARSENL swarm system. The following terminology is used throughout:

- **Behavior:** A controller that is loaded into individual UAVs *Swarm Control* nodes to effect specific swarm-wide performance.
- **Waypoint:** A geographic coordinate to which the robot is to travel.
- **Waypoint Command Message:** Within the *Swarm Control* node a *WaypointCommand* is a container object in which waypoint information is maintained for publication to the *Autopilot* node.
- **Waypoint Sequence:** An ordered list of waypoints derived for the purpose of directing UAV travel through a specific flight pattern (derivation is described in Section 3.4.2).
- **Play:** A specific type of behavior that combines multiple algorithms to control swarm UAVs in the completion of a specific task. The most important function of a play is the generation of waypoint commands in each control loop iteration to direct the vehicle through completion of the play's task.
- **Resource:** A task (e.g., a *search cell* or *contact investigation*) that is included in the the auction for assignment to a swarm UAV.

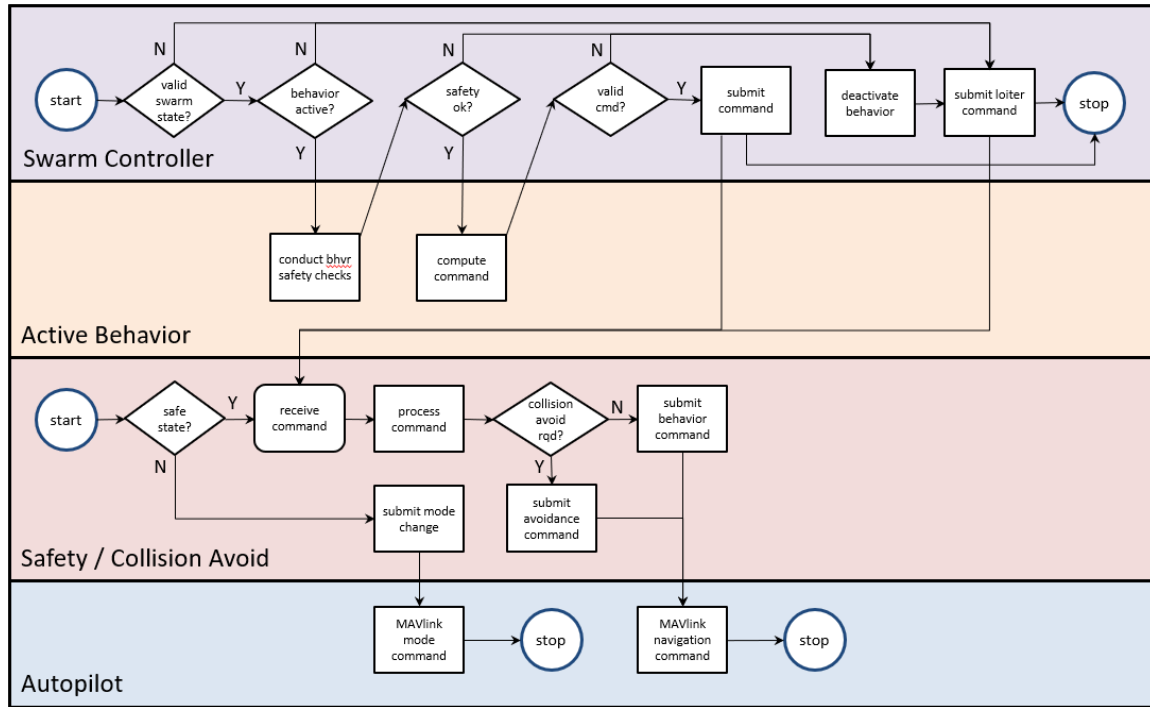


Figure 3.2. The *Swarm Controller* node uses the active plugin behavior to generate commands for relay to the *Autopilot* node at a rate of 10 hertz.

- **Resource Record:** The ResourceRecord class is used to consolidate information about a single auction resource.
- **Search Cell:** The search area is discretized into smaller search cell tasks as SearchCell objects that are included in the auction as biddable resources. The initial auction described here consists solely of search cell tasks.
- **Investigation Point:** Contact detection events that occur over the course of the area search generate ContactInvestigation objects that are included in the auction as biddable resources. When a robot identifies a contact requiring investigation, a *contact investigation* task is generated and a new auction is immediately commenced to account for the new resource.

3.2.1 Cued Search Behavior Design

The `auction_searcher_test` play is comprised of three main components that execute the autonomous cueing search behavior. The first component of the architecture defines the search space and partitions it into smaller search cell resources that are incorporated into the

auction as `SearchCell` objects. Specific predetermined cells are designated as “detection” cells when they are generated to facilitate testing of the cueing functionality. When a robot searches one of these cells, it will find a contact and instantiate a `ContactInvestigation` object for incorporation into the next auction. These resources and any other subsequent tasks are defined in a separate `searcher` module which allows the swarm to deal with them abstractly as generic resources in the auction. Section 3.4 describes the mechanism that allows the definition of different types of tasks as children or augmented replicas of the existing task classes. The behaviors associated with those tasks follow a simple structural framework that defines the specific trajectory of a robot assigned to the task.

The second component of the architecture is the auction which is an implementation of Bertsekas’ auction we described in Chapter 2. Each robot instantiates a `SingleItemAuction` object, as depicted in Figure 3.3, which handles the bidding and assignment process using abstract resources that are defined in the `searcher` module. Since the swarm is fully distributed, each robot’s `SingleItemAuction` object updates the associated values independently based on the communications it exchanges with the other robots. Section 3.5 describes how we implemented the auction algorithm and the data structures used for maintaining consistent bids across the UAV swarm.

Lastly, the `SearchCell` and `ContactInvestigation` objects define specific search and investigation tasks, respectively, that the robot must execute to complete the task if it is assigned. As each robot is assigned a task, the robot’s `AuctionCue` behavior’s *computeCommand* method unwraps the details of the task and invokes the appropriate `Driver` (depicted in Figure 3.3) object to compute the command. If the robot is assigned a search task, a `WaypointSequenceDriver` object is used to drive through a search pattern. If, on the other hand, the robot is assigned to an investigation task, a `DirectDriver` object is used to direct the vehicle directly to the contact location. These objects and their derivation are described in detail in Section 3.4.

The following sections demonstrate how this architecture serves as a design blueprint for our implementation as well as any expansion to account for more types of tasks and greater heterogeneity.

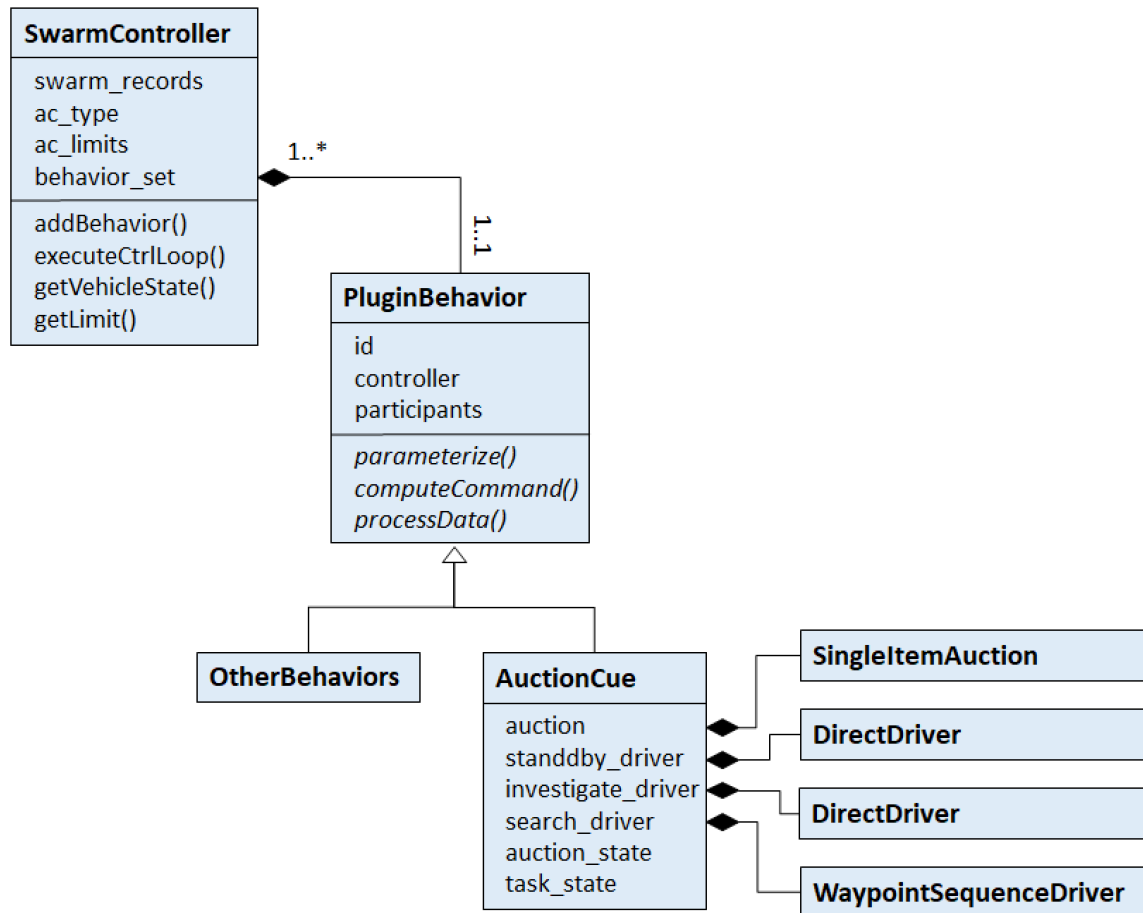


Figure 3.3. The `AuctionCue` class inherits from the abstract `PluginBehavior` class. An instantiated `AuctionCue` object is loaded into the `SwarmController` object at start up along with other behaviors. The object incorporates a `SingleItemAuction` object and multiple `Driver` objects.

3.3 Behavior Class

The robot behavior is loaded into the *SwarmControl* node as an `AuctionCue` object, as depicted in Figure 3.3. The `AuctionCue` class inherits from the abstract `PluginBehavior` class that provides access to the functionality of the `SwarmController` object into which it is loaded and a list of other swarm UAVs participating in the behavior. The `AuctionCue` class implements the `PluginBehavior` class' abstract *parameterize*, *computeCommand*, and *processData* methods.

When active, the behavior's *computeCommand* method is invoked by the *SwarmController*

node at a rate of 10 hertz as depicted in Figure 3.2. The cued search behavior control flow implemented in this method is depicted in Figure 3.4.

Within the *computeCommand* method, the robot maintains two state variables: an auction state (*auction_state*) and a task state (*task_state*). The auction state is either *RESTART*, *IN_PROGRESS*, *INACTIVE*, or *COMPLETE*. The task state is either *SEARCH*, *SEARCH_WAIT*, *IDLE*, or *INVESTIGATE*. In any extension of this code the number of auction states will remain the same for a single item auction, but the number of task states can grow arbitrarily to account for additional task and vehicle heterogeneity. Maintaining two state types enables robots to participate in an auction while simultaneously executing previously assigned tasks, facilitating their parallel execution.

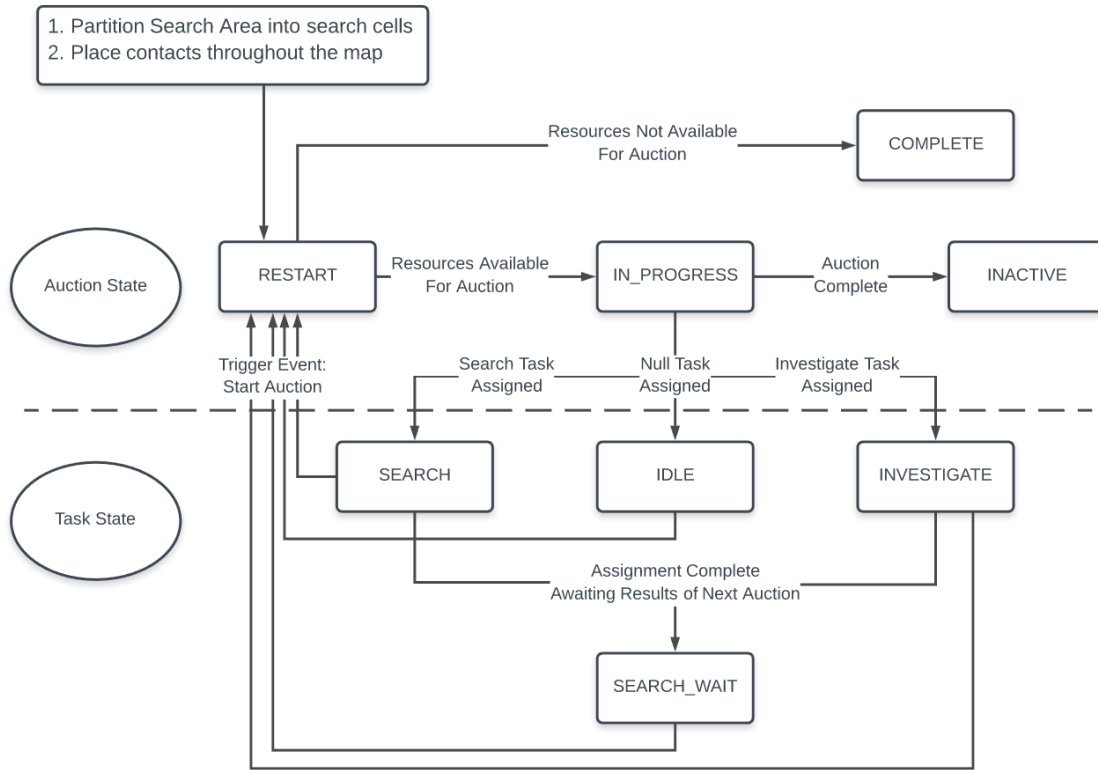


Figure 3.4. Each robot maintains two states: an auction state and a task state. The auction state maintains the state of the auction for the robot swarm while the task state maintains the status of the current task that an individual robot is assigned. Two states allow the robots to participate in auctions and execute their assigned tasks simultaneously.

Upon activation, the play is parameterized with the area to be searched, and the search area is partitioned into search cell resources as part of the initialization process. The search cells that are to contain contacts are also defined at this time but the actual contact locations remain hidden from the UAVs until the cell is searched and the contact is “discovered”. All robots’ auction and task states are initially set to *RESTART* and *SEARCH_WAIT* respectively.

Once the resources have been properly instantiated, the first auction commences and the auction state transitions to *IN_PROGRESS*. Upon completion of the initial auction, the UAVs commence execution of their assigned tasks, and the auction state transitions to *INACTIVE*.

Upon commencement of an assigned task, a robot unpackages the task identification and parameters and computes the trajectory requirements associated with the task. The task identification dictates the type of task and therefore the updated task state. Following the initial auction, all robots are assigned search tasks, so all robots’ task states transition to *SEARCH*. Subsequent auctions, however, can include various types of tasks, each of which will define its own type-specific task states. All the robots carry out their assigned tasks and the auction will only reactivate if a triggering event sets the auction state to *RESTART*.

In our implementation, there are two events that can trigger a new auction. The first is when a robot finishes its currently assigned task and has no tasks left in its assignment queue. The second is when a robot identifies an investigation task that must be bid on immediately. All robots participate in all auctions and are assigned a new task (which may be *null*) upon the completion of each one. A newly assigned task will not interrupt a currently executing one but will replace a previously assigned “next” task if there is one. The bid exchange process is described in Section 3.5, and the utility function described in Section 3.6 is used to derive local utility values for all available tasks.

3.4 Auctioned Task Objects

In order to maintain consistency, task objects were built within a standardized framework. First, all task objects require a *taskID* attribute. The *taskID* labels the task object and indicates what type of task it represents. This is important because task states, their transitions, and the events upon which those transitions take place can vary between task types.

There are currently two types of task objects that can be assigned to a robot as the result of an auction; search cells and investigation tasks. A search cell is a task that requires the vehicle to systematically traverse a coverage pattern (i.e., search the area defined by the cell). The coverage pattern is defined as a sequence of latitude/longitude/altitude waypoints that is used by a `WaypointSequenceDriver` object to control vehicle travel during the search. A contact investigation task, on the other hand, is a task that requires the vehicle to expeditiously transit to a geographic location and loiter there for 20 seconds to “investigate” the contact. This type of task uses a `DirectDriver` object to control the transit and loiter process.

3.4.1 Search Area Discretization

The `auction_searcher_test` play’s *parameterize* method initializes all of the attributes that are required for the code execution. The most important initialization requirement is the definition of the search area and the discretization of the area into search cells. We have adopted the cell partitioning code from [10] into our implementation. Thus, our single item auction loosely replicates the behaviors observed in that work when run with a homogeneous swarm with no contact generations. Cells are partitioned from the search area when the *parameterize* method is called at runtime. For this thesis, we utilized two search areas: a basic search area suitable for the live-fly events and a large search area for experimentation and data collection in the simulation environment. Both areas are rectangular and are defined using the latitude/longitude of their southwest corners, the north/south and east/west lengths of the rectangles, and the clockwise rotation of the rectangle as depicted in Figures 3.5 and 3.6.

As implemented, the cell partitioning method standardizes the boundaries and *cellIDs* of all search cells among the participants so that auction consistency is maintained. It also ensures the creation of `SearchCell` task objects which have attributes and methods common to all task objects that facilitate task execution. `SearchCell` tasks are explained further in Section 3.4.2.

3.4.2 Search Cell Tasks

Search cell objects are instantiated as resources and exchanged in the auction market as described in Section 3.5. These cell objects are encapsulated within `SearchCell` tasks

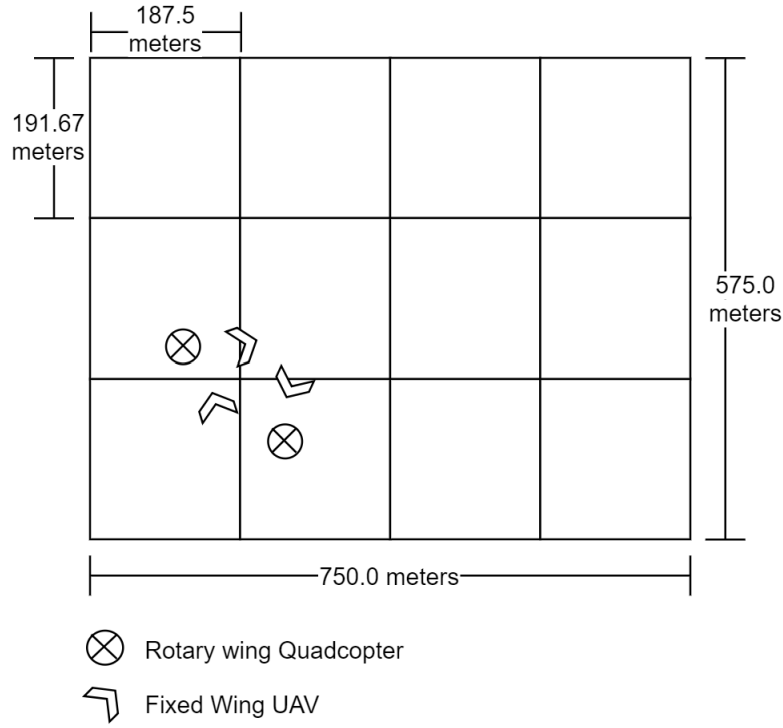


Figure 3.5. The basic search area partitions a 750-meter by 575-meter search area into 12 search cells containing two hidden contacts.

that require full coverage for completion. Each cell has an intrinsic value based on the contribution its completion will make to the overall search. That is a function of the size of the cell and is maintained as a *taskValue* attribute. The intrinsic value of a cell is computed as:

$$v_i = l_i + w_i - 130.0 \quad (3.1)$$

where v_i is the value of cell i and l_i and w_i are the length and width respectively. The cell value formula has been manipulated to produce a per-cell value of 250 for the basic search area. In particular, basing the cell's value on its dimensions rather than its area causes the task value to increase linearly vice quadratically so as to avoid the overvaluation of slightly larger tasks. Although our implementation divided the search areas into uniformly sized cells, Equation 3.1 supports discretization into arbitrarily sized cells as well. Larger cells

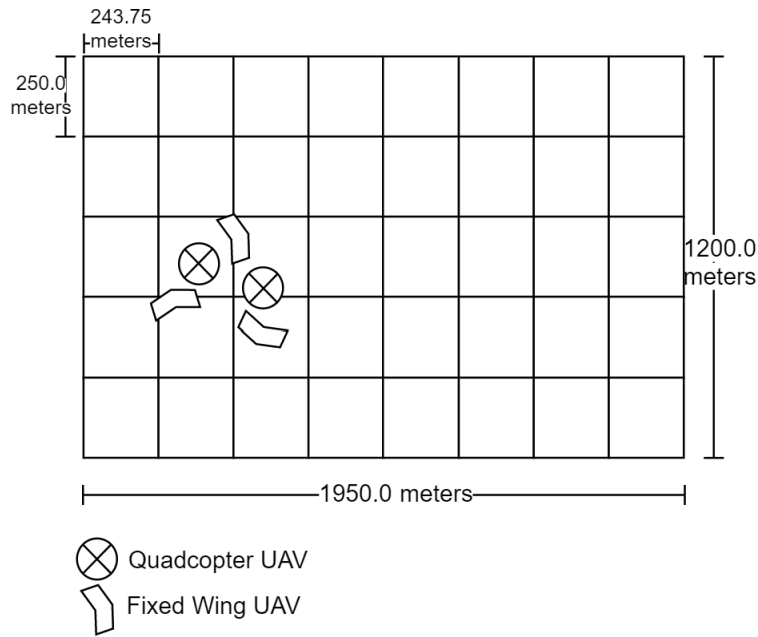


Figure 3.6. The large search area partitions a 1950-meter by 1200-meter search area into 40 search cells containing 10 hidden contacts.

would be assigned preferential value over smaller cells in these cases. Once the search area is discretized and values assigned, the task value and location of the cell remains fixed throughout the search, and provide all of the information required for evaluation over the course of multiple auctions. This process is described in detail in Section 3.5.

Upon commencement of a search cell task, a robot must generate a path to follow in order to carry out an exhaustive search of the region. That is, the task's `WaypointSequenceDriver` must be initialized with a waypoint sequence for the search definition that accounts for the current location of the robot. For example, if the robot is transiting to the assigned cell from the southwest, the waypoint sequence must begin with the southwest corner of the cell. Next, a series of latitude, longitude, altitude, and speed waypoints are appended to the sequence to efficiently cover the cell. Because our search cells are rectangular, a lawnmower pattern as depicted in Figure 3.7 is utilized.

For testing purposes, we arbitrarily chose a sensor sweep width (i.e., the distance from the vehicle visible to the sensor) of 25 meters and assumed a 360-degree field of view. Generated search patterns, therefore, have a track spacing (i.e., distance between parallel

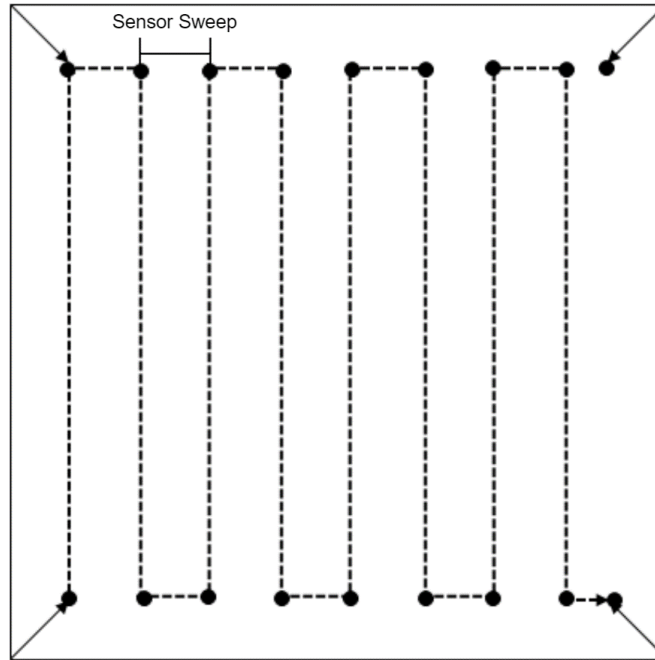


Figure 3.7. Waypoints are arranged in a lawn mower pattern for search cell coverage with the first waypoint located nearest the corner by which the UAV will enter the area.

search legs) of 50 meters. This value can be adjusted as appropriate to account for different sensor characteristics.

Waypoints are sequenced starting at the closest corner to the UAV's current location. The pattern is generated as a series of uniformly spaced parallel tracks. The cell entry corner is connected to the first track's starting point, and each track's ending point is then connected to the subsequent track's starting point until the lawnmower pattern is complete. The completed sequence is used to initialize the `WaypointSequenceDriver` that will control the robot through the search.

3.4.3 Investigation Tasks

While the robots are executing search tasks, they may encounter contacts which require further investigation. During these detection events, the UAV that has identified the contact notes its latitude and longitude and relays the contact report to the rest of the swarm for in-

clusion in the next auction as an investigation task. Investigation tasks are assigned a higher intrinsic value than the search cells so that their completion will be prioritized. Further, identification of a contact will immediately initiate a new auction to ensure their investigations are completed expeditiously. Ideally, investigations are carried out by quadrotors UAV because the quadrotor is the most suited platform for this type of task. In the navy, this event is analogous to a fixed-wing aircraft such as a P-8 conducting a search and cueing an MH-60R to investigate a possible target of interest.

Contacts do not exist in the auction at the outset of the play since they are triggered by asynchronous detection events as robots discover contacts in their assigned search cells. Since this thesis is not concerned with the actual detection of contacts, we intentionally inject these events into the search of predetermined cells. These cells are determined upon play commencement by deterministically identifying 25 percent of the cells and designating them as “detection” cells. When a robot searches one of these cells, the detection event is randomly generated (no later than the 65 percent completion). A robot that detects a contact immediately instantiates a `ContactInvestigation` object and alerts the other robots by sending a `ContactReport` message. Robot communications requirements are described in depth in Section 3.7.

While the task value of a search cell varies with the size of the cell, the task value of a `ContactInvestigation` is fixed to a value that reflects prioritization of contact investigations relative to searches. In order to ensure prioritization of investigation tasks over search cell tasks, we set the investigation task value to 600.0 for larger search area and 350.0 for the smaller area.

Since contact’s location is set to that of the detecting UAV and local utility values factor in required transit time, the detecting robot is obviously the closest to the contact at the time of the auction. It might seem like this gives the discovering robot has an inherent bidding advantage; however, the underlying utility function discussed in Section 3.6 is implemented so that this advantage rarely manifests itself.

3.5 Single Item Auction Implementation

The auctions process abstracts the task object as a resource. Every resource is representative of either a `ContactInvestigation` or a `SearchCell` object. Hypothetically a resource

can be any task object if we were to define additional tasks in a more heterogeneous assignment problem. The abstraction approach allows for task scalability and promotes flexibility in the variability of tasks. While the robots are conducting the auction, the type of resource drives how they make their valuation but the framework that accepts and exchanges bids is built solely to make an assignment of a robot to an abstract resource.

3.5.1 The Single Item Auction Architecture

The auction framework is implemented by the `SingleItemAuction` class as depicted in Figure 3.8. Each robot instantiates its own instance of the `SingleItemAuction` when play execution is initiated. As stated previously, the only resources that the robots can initially bid on are the search cell tasks. Contact investigation tasks are incorporated into the auction upon detection events as discussed in Section 3.4.3. The `SingleItemAuction` maintains or provides access to all information and functionality required by the auction algorithm including the identifications of auction participants, vehicle state and capability information, and the bid exchange. When this instance is initialized, an empty `ResourceSet` object is also instantiated to maintain the search and investigation tasks as generic `ResourceRecord` objects that are bid upon in the single-item auction.

3.5.2 The Resource Set

The resource set is a collection of resources (`ResourceRecord` objects) available for allocation via the auction process that correlates those resources to unique identifiers. When the behavior is initialized, first the search area is defined and partitioned as described in Section 3.4.2. Simultaneously, a `ResourceRecord` object is instantiated for each cell and added to the robot's resource set. Since the search area discretization is deterministic, the resources are defined consistently across the swarm.

The resource record is used by each robot to consolidate information about the auction resource including its availability status, the local utility value, and the current bid for the resource. As the auction progresses and robots make their bids, these attributes are manipulated by the individual robots so that each maintains its own record of the auction status.

Adding a contact investigation tasks or any other task types is fairly straightforward. Since

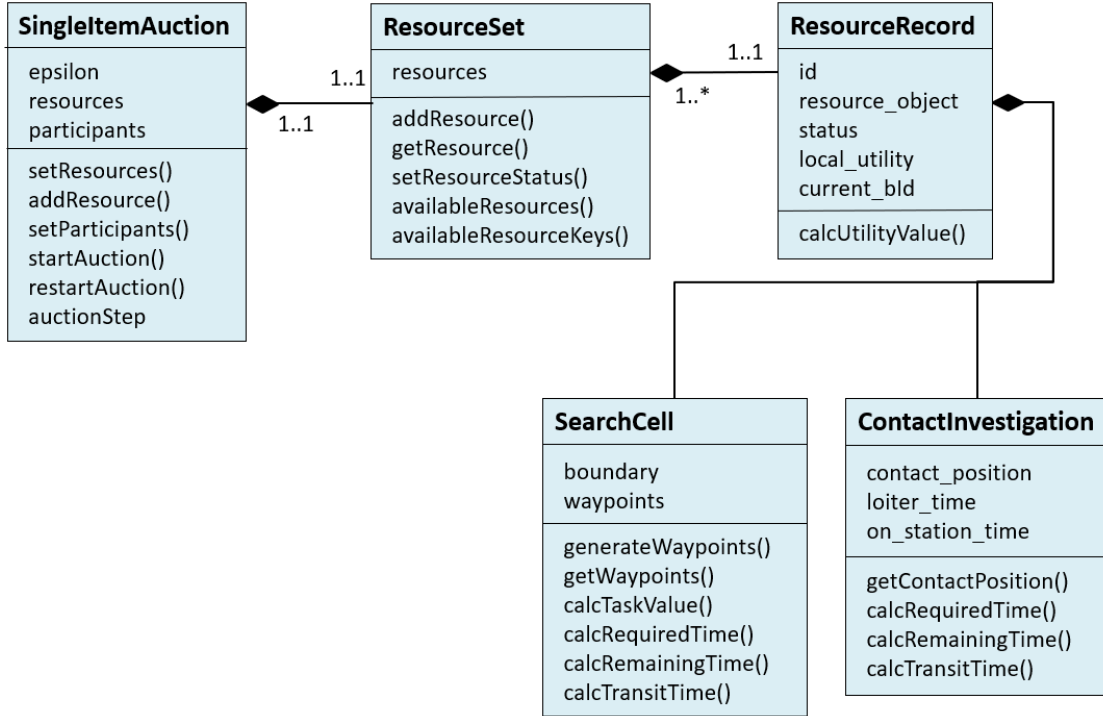


Figure 3.8. The `SingleItemAuction` class implements Algorithm 2. Auction resources are maintained in a `ResourceSet` object containing `ResourceRecord` objects defining `SearchCell` and `ContactInvestigation` tasks.

all task objects are incorporated into the auction as generic resources, all that was required for cued investigation task inclusion was development of the `ContactInvestigation` class implementing the investigation-specific functionality. Objects of this class were then directly incorporated into the auction in `ResourceRecord` objects. In other words, the architecture of our auction algorithm codebase facilitates a flexible heterogeneous task implementation.

3.5.3 Auction Execution and Task Assignment

At the start of every auction, including the initial search cell assignment auction, the auction state is set to *IN_PROGRESS*, and the the resource set is updated to reflect the currently available resource states (e.g., current bid prices are reset to zero) and the auction's round is set to zero. No-task (i.e., *null*) resources are added to the resource set as required at this time to ensure at least as many tasks as robots. These resources have no value but facilitate

auction termination when there are more robots than currently available resources per the problem outlined in Section 2.2.2.

Auction iterations (rounds) are managed by the play's *computeCommand* method. The following steps are executed in order for each round until the auction is complete:

1. *Generate Bid*

- (a) This step determines the robot's desired resource and bid price to be posted for the current round. The bid consists of the round number, the resource identification, and the bid price. Bids from all participants are consolidated and processed in steps 2 and 3.
- (b) The robot transmits its bid to the other robots.

2. *Synchronize and Update*

- (a) Bid information for the current and previous round is retransmitted to swarm robots as required to account for the ARSENL swarm system's lossy communications architecture.
- (b) The robot determines whether or not all bids for the current round have been received. If not, a request for missing bids is transmitted and the step is paused and then repeated until all bids have been received. This ensures synchronization by not allowing a robot to proceed to the next round prematurely.
- (c) Once all bids have been collected for the current round, the *ResourceRecord* object for each resource that was bid upon is updated with the highest bid for the current round and associated with the submitting robot.

3. *Auction Complete*

- (a) Auction completion is determined by whether or not every participating robot is associated with a resource. If this is the case, then no two robots are competing for the same resource and they will all be "happy". If this is not the case, then at least one robot was outbid for its chosen resource.
- (b) If all the robots have an associated with a resource, return the computed assignments and terminate the auction. If not, increment the auction round and continue.

During each iteration of the play's control loop, this verification and update process is executed until the auction results are available. In practice, the process resolves fairly

quickly, returning assignments within a reasonable amount of time so that the robots can execute their assigned task. Auction algorithm performance is discussed in Chapter 4.

Each robot maintains a record of up to two task assignments: the *currentTask* assignment and the *nextTask* assignment. When a robot is assigned a task via the auction process, it is always saved as the *nextTask*. When the robot has an empty *currentTask*, the robot will move the *nextTask* value to the *currentTask* record. If a robot finishes its *currentTask* and the *nextTask* is empty, a new auction is initiated if one is not already in progress. A task is not made unavailable to the auction until its execution commences, so tasks that are maintained in a robot's *nextTask* record remain available. In other words, the resource is still available for bids and can be “stolen” by another robot until the currently assigned robot actually starts the task.

The *nextTask* place-holder enables a robot to participate in an auction prior to completing its current task. Tasks that are assigned to the robot's *nextTask* record are only notionally assigned to the robot and are available for subsequent auctions. If a robot completes a task and the *nextTask* record contains a previously assigned task, the robot will commence execution of that task rather than initiating a new auction. At the same time, the robot will notify the other swarm UAVs that the task is being executed so that it will be made unavailable in future auctions. For example, in Figure 3.9, if robot 4 finishes task 14 before robot 3 finishes task 0, robot 4 will initiate an auction. Since robot 3 has not commenced task 3 execution, it will still be available for bid and robot 3 could be assigned another task. On the other hand, if robot 3 finishes its current task (task 0) first, it will continue on to task 3 without initiating an auction. In this scenario, task 3 will become *ACTIVE* and no longer available for auction.

3.6 Utility Function and Task Valuation

3.6.1 Time-Based Local Value Calculation

As described in Section 2.4.1, each robot maintains a local utility value for each available resource and bids accordingly using Equation 2.10. The local utility function is typically implemented as the difference between the resource's intrinsic value and a robot-specific cost associated with its completion. The intrinsic value of completing a task without cost is

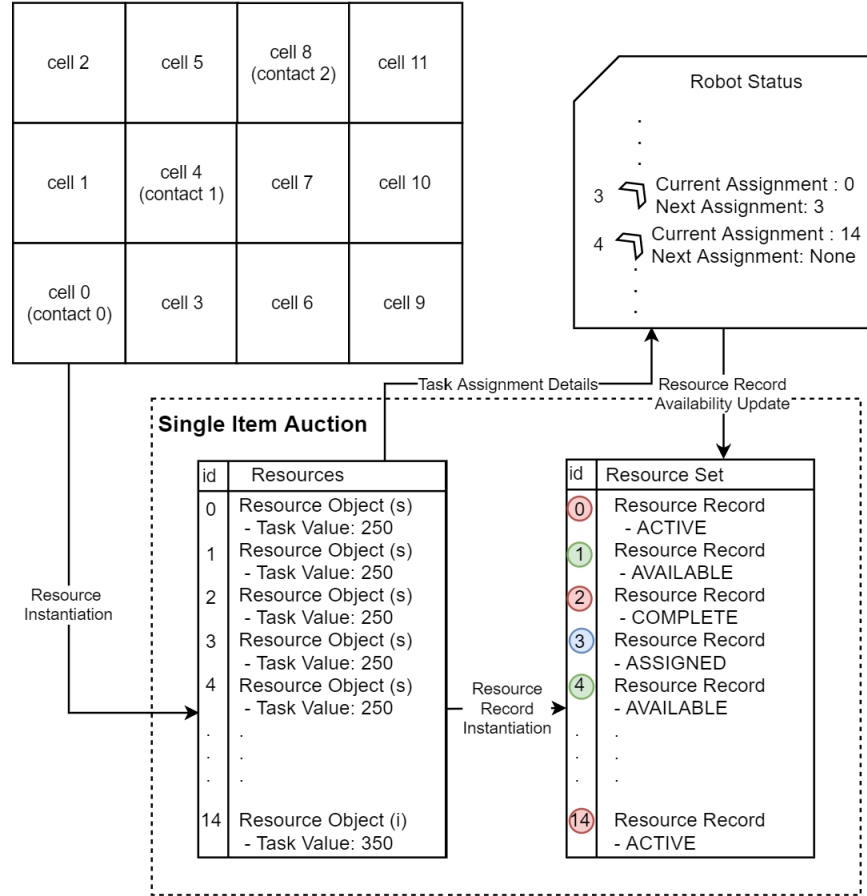


Figure 3.9. Search and contact investigation tasks are encoded as generic ResourceRecord objects when they are added to the auction. A ResourceSet object maintains the availability of the tasks. Robots exchange bids according to the locally computed utilities and current bid prices of the tasks. A task remains *AVAILABLE* until a robot commences its execution and resets its status to *ACTIVE*.

maintained within each task object with search and investigation task values set as described in Section 3.4. The vehicle-specific cost associated with a task's completion is reasonably described as a function of the time required for completion for both search and investigation tasks. The total time required is the sum of the amount of time required to complete the robot's current assignment (or zero if the robot is not currently assigned to a task), the time required to transit to the next assignment, and the time required to complete the task once commenced. The total time upon which robot i currently executing task j would base its

local cost to execute task k as Equation 3.2 where $t_{j_{remain}}$ is the remaining time to completion of task j , $t_{jk_{transit}}$ is the time required to transit from task j to task k , and t_k is the time required to complete task k .

$$t_{ik} = t_{j_{remain}} + t_{jk_{transit}} + t_k \quad (3.2)$$

The first component of the time calculation is the expected time remaining to complete the task to which the robot is currently assigned (if there is one). If the robot is currently assigned to a search task, the time remaining to completion is calculated using the time-speed-distance formula ($time = distance/speed$) to determine time required to transit between each of the remaining search pattern waypoints. If, on the other hand, the robot is assigned to an investigation task, the expected time remaining is simply any remaining loiter time (i.e., $arrival\ time + loiter\ time - current\ time$). If the robot is not currently assigned to a task, the time remaining is zero (e.g., before the first auction or when a robot does not have another assignment in its queue).

The second component of Equation 3.2 is the time it takes for the robot to transit from the last waypoint of its current task to the first point of the task being evaluated. If the currently executing task is a search task, the starting point of the transit will be the location of the final waypoint in the search pattern. If the currently executing task is a contact investigation, the starting point will be the contact location. Similarly, the end point of the evaluated task will be the search cell corner closest to the transit starting point for a search task or the contact location for an investigation task. Evidently, tasks that will commence their transit from closer to the evaluated task's location will incur a smaller penalty because it requires less time for the robot to travel to them.

The third component of Equation 3.2 is the anticipated time it will take for the robot to accomplish the evaluated task. For search cell tasks, the time-speed-distance formula can be used to determine the amount of time the vehicle will require to complete the search pattern. For contact investigation tasks, completion time is computed simply as the required loiter time. Because they are slower, quadrotor UAVs can be expected to devalue search tasks in general and large search cells in particular relative to fixed-wing UAVs. The task completion component of the required time calculation does not favor either platform.

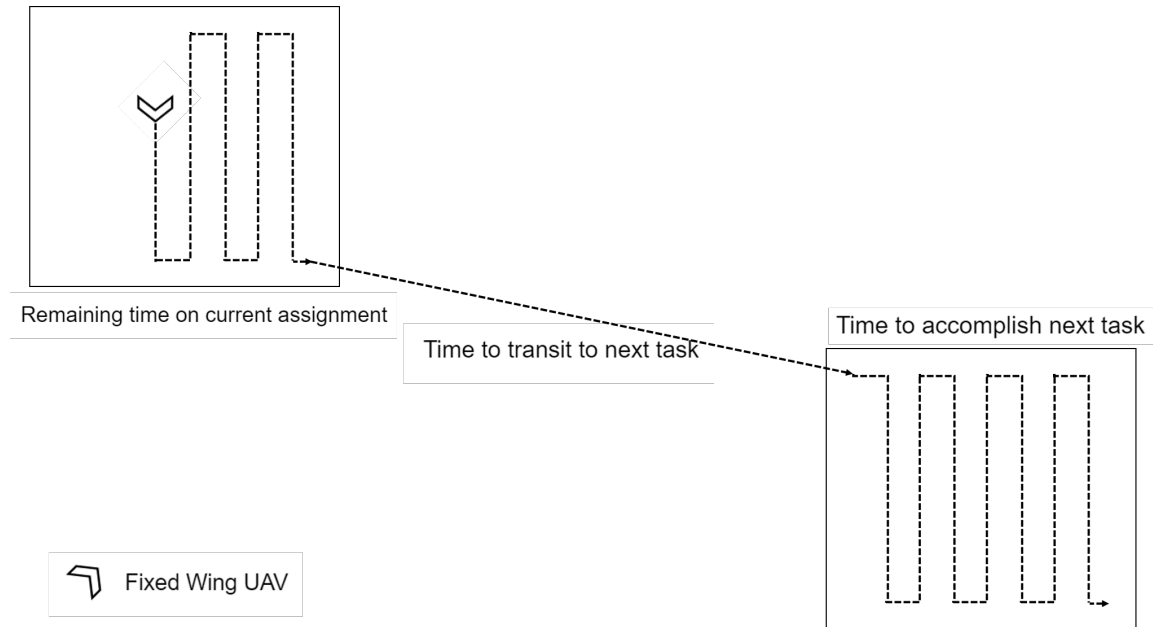


Figure 3.10. Each UAV computes the local utility value of a task based on the amount of time required to complete their current assignment, the anticipated transit time to the next task, and the time required for the UAV to complete the evaluated task.

3.6.2 Airframe Consideration

In a heterogeneous swarm, we want to see tasks paired with the robot type most appropriately equipped to execute the task. In our implementation, this means that we want to have the fixed wing UAVs bidding more aggressively for search tasks (particularly those that are more distant or larger) and the rotary wing aircraft bidding more aggressively for investigation tasks. Ideally, this can be accomplished by basing local utilities purely on the required time to complete the various tasks. In practice, using raw required execution time did not provide enough differentiation to incentivize the quadrotor and fixed wing UAVs to respectively value investigation or search tasks more highly. Furthermore, our original premise was based on the understanding that certain types of vehicles would be more suitable than others to certain types of tasks (Figure 3.11). Basing the utility value solely on aircraft speed will not account for these differences.

In order to account for these capability differences, the local utility function must include aircraft type as a parameter. We accomplished this by introducing an airframe and task-specific coefficient to bias the utility towards the more desirable airframe type. The

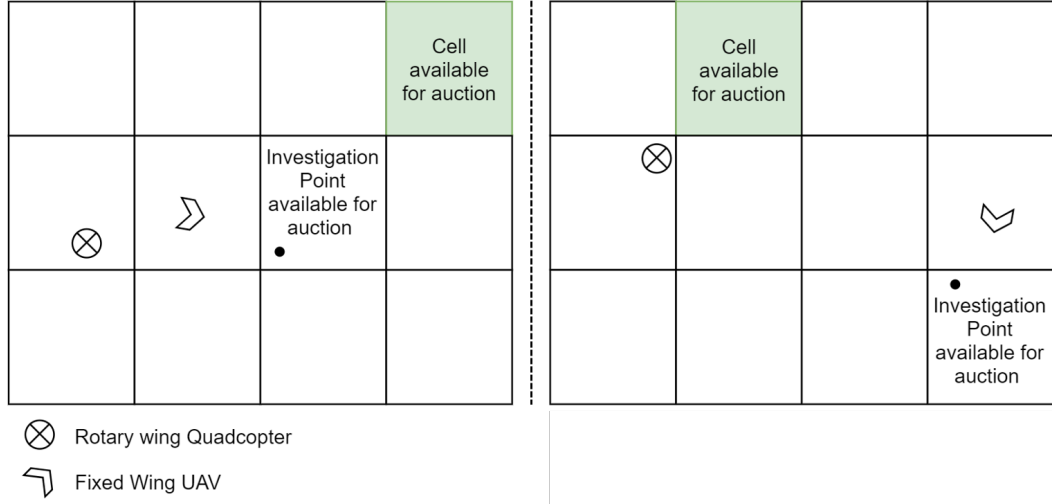


Figure 3.11. Depicted are two scenarios in which fixed wing and rotary wing UAVs are idle while participating in a task assignment auction. In both cases, the fixed wing UAV is closer to the investigation task for which it is less well suited. The utility function must impose an additional cost to the fixed wing UAV for choosing an investigation point if it is to elicit “preference” for the search cell tasks. The cost should not be so punitive that an aircraft would always prefer the most appropriate task, even when extreme proximity differences exist as in the second scenario.

coefficient effectively penalizes utility calculations for tasks for which the evaluating robot is ill-suited. The airframe specific local cost associated with robot i ’s completion of task j , therefore, is given by Equation 3.3:

$$cost_{ij} = C_{ij} * t_{ij} \quad (3.3)$$

where C_{ij} is the airframe-specific multiple associated with robot i ’s assignment to task j and t_{ij} is computed with Equation 3.2.

The airframe-specific coefficients were manipulated to ensure that quadrotors consistently valued investigation tasks higher than search tasks and fixed-wing UAVs consistently valued search tasks more highly. For contact investigation tasks we obtained reasonable performance with a coefficient of 2.0 for quadrotor UAVs and 1.0 for fixed-wing UAVs. For search cell tasks, our coefficients were set to 1.0 for quadrotor UAVs and 2.0 for fixed-wing UAVs.

3.6.3 Final Utility Value Calculation

Per Section 3.4 contact investigation tasks are intrinsically considered more valuable than search tasks to encourage robots to execute them as soon as possible regardless of airframe type. Combining these values with Equation 3.3 yields the full local utility function for UAV i for task j of Equation 3.4.

$$u_{ij} = v_j - cost_{ij} \quad (3.4)$$

The main consideration in determining the utility function was ensuring that search cells and contact investigation tasks were valued appropriately to encourage behaviors we would expect from a human-controlled aerial search system consisting of vehicles with different characteristics and capabilities as discussed in Section 3.6.2. We, therefore, want robots to place higher values on tasks for which they are most suited and that will require less time to complete (i.e., tasks that are closer and more expeditiously accomplished). Experimental results described in Chapter 4 empirically support the suitability of the local utility function described here.

3.7 Intra-Swarm Messaging and Notifications

Swarm robots rely on intra-swarm communications to carry out their tasks, exchange auction information, and notify each other of resource statuses. There are three notification messages that are exchanged from within the play: a new auction initiation, a contact detection alert that requires investigation (which implicitly initiates a new auction), and update to the status of a resource. The communications framework allowed the definition of additional notification messages as required to support additional heterogeneous tasks. In addition, the auction messaging framework itself can be extended to support additional market-based implementations for distributed task assignment.

The most frequently utilized notification message is the *notify task status* message. Task statuses are maintained in the `ResourceRecord` class discussed in Section 3.5. When a robot enters an auction, it must know which resources are available for bid and which are not. The robot notifies other robots of the change in status with a *notify task status* message containing the task's *resource identification* and the updated status.

New auctions are initiated when a robot requires a new task assignment, and it does not have one in its cue. Robots use a *new auction* message to notify robots that a new auction is starting. Since the auction will only include currently available tasks, this is an empty message and no data is actually exchanged.

New auctions are also initiated by contact detection events, but more information has to be passed before the auction starts. The new investigation resource must be added to the auction and the other robots need to be informed of the contact's location. A *contactReport* message is used to both notify other swarm UAVs of the new contact and its location and also initiate a new auction that will include the associated investigation task.

3.8 Summary

In this chapter, we discussed our implementation of Bertsekas' auction algorithm in a heterogeneous UAV swarm. By starting with a simple implementation of two task types, two types of robots, and a framework for auctioning the tasks among participants in a robot swarm, we built a system that can scale to an implementation involving more than two task types and two types of robots. Our architecture was developed in an object-oriented manner that will allow for a straightforward expansion to incorporate increasingly heterogeneous tasks and vehicle types. In Chapter 4, we discuss results observed in simulation and live-fly experimentation with this system.

CHAPTER 4: Experimentation

This chapter summarizes the results obtained during experimentation. Experiments were conducted in simulation and live flight to assess the play’s performance in the generalized assignment problem described in Chapter 2. In particular, we explore a scenario in which the heterogeneous swarm comprised of the UAV types depicted in Figure 4.1 conducts an area search that is interrupted by detection events requiring investigation by a member of the swarm. As the search progressed, robots that detected potential targets of interest independently generated contact reports that initiated new auctions. The new auctions included both the remaining search cell tasks and the target investigation tasks. Furthermore, experiments were conducted with a heterogeneous swarm consisting of both fixed-wing and quadrotor UAVs. This heterogeneity led to a swarm where individual UAVs were more suitable to either the search or investigation task. Our results demonstrate that the proposed auction algorithm is an effective way to optimally dynamically assign members of this heterogeneous robot swarm to heterogeneous tasks as those tasks are discovered or changed.

4.1 ARSENL Simulation Environment Experimentation

As with Hopchak’s work [10] our `auction_searcher_test` experimentation relied heavily on the ARSENL Software in the Loop (SITL) simulation environment. This environment was utilized to test and develop the implementation, eliminate bugs in the behavior, and to collect data supporting performance analysis. The SITL capability provides a realistic environment in which actual on-board software and communications architectures can be rigorously tested [29]. This environment allowed us to perform simulations of the entire UAS as it would run on actual robots in the field. In addition, the SITL simulation allowed us to run experiments that were unsuitable for live-flight testing (i.e., they produce potentially unsafe conditions or require flight outside of the approved flight area). The simulations also helped us evaluate performance in various edge cases to facilitate algorithmic robustness. Two live-fly events were conducted at McMillan Airfield, Camp Roberts CA over the course of this work to validate SITL simulation results.



Figure 4.1. The two UAV airframes that we used in our implementation: the Zephyr II fixed-wing is depicted in the top image and the Mosquito Hawk quadrotor is depicted in the bottom image.

In total, we conducted SITL environment experiments with eight different swarm sizes ranging from three robots to ten robots. Even numbered swarms were composed of an equal number of quadrotors and fixed-wing UAVs, while odd numbered swarms incorporated an additional fixed-wing UAV. The number of fixed-wing and quadrotor UAVs in the experimentation robot swarms was kept as even as possible in order to minimize bias in the assignment of robots to tasks. Data presented in Section 4.3 was collected over a minimum of ten SITL simulation experiments for each swarm configuration.

4.2 Expected Behavior

We expected to develop a UAS that could autonomously determine optimal task assignments in order to exhaustively search an area and investigate every contact. Our primary goal was to ensure that robots that were best suited to tasks bid partially to those tasks. We expected that when there were quadrotor UAVs available, they would win contact investigation tasks. As a corollary, we expected that the robot would conduct the search in an organized manner, starting with the closest cells and systematically searching the defined area. We wanted

discovery events to be a relatively seamless interruption of the search. Our framework outlined in Chapter 2 and the simultaneously manipulated *Task State* and *Auction State* was built specifically to ensure minimal disruption from the cueing behavior.

Among our standards for a successful implementation was one that minimized the amount of time a robot was waiting for a task assignment or transiting between tasks. Keeping both wait and transit time low maximizes the robot utilization, measured as the proportion of time the robot spends actively searching or investigating relative to the total time it takes to complete the search. This ratio is an intuitive proxy for the measure of the overall performance of the auction algorithm in a search environment because it encompasses how much time the robot is idle, a reflection of the computational efficiency of the algorithm and also the time it takes robots to transit to their assigned task, a reflection of the optimality of the task assignment. Idle periods where robots are waiting for the results of an auction and unnecessarily long transits result in lower utilization ratios. An idealized case in which the robot utilization of all UAVs is 1.0 (i.e., all robots always actively engaged in their tasks) provides a theoretical “best case” against which test results can be compared. We were satisfied with robot utilization above 0.5 as an indicator that our algorithm was finding near optimal assignments of robots to tasks.

4.2.1 Preliminary Observations

At the start of the play’s execution, all the robots loitered at a standby point as shown in Figure 4.2. In fact, the fixed-wing aircraft are slightly closer to the search cells because their flight characteristics do not allow them to hover in one place. Given our defined utility function based on speed and UAV airframe type, we expected to see the rotary wing quadrotors assigned to the closest search cells and the fixed-wing aircraft assigned to the cells just beyond the nearest search cells. Though the fixed-wing aircraft can reach the closer search cells and complete them faster than the quadrotors, the quadrotors are more averse to being assigned search cells farther away because of their slower speed relative to the fixed-wing UAVs. Therefore they bid more aggressively for the nearest cells. As the auction progressed we expected this trend to continue with quadrotor UAVs winning the closest cells in most cases. This expected behavior is shown in Figure 4.3 where the fixed-wing UAVs are searching the cells furthest from the standby position. As the play progressed we observed this inherent spreading of the swarm as the robots were assigned

tasks according to the utility function we defined.

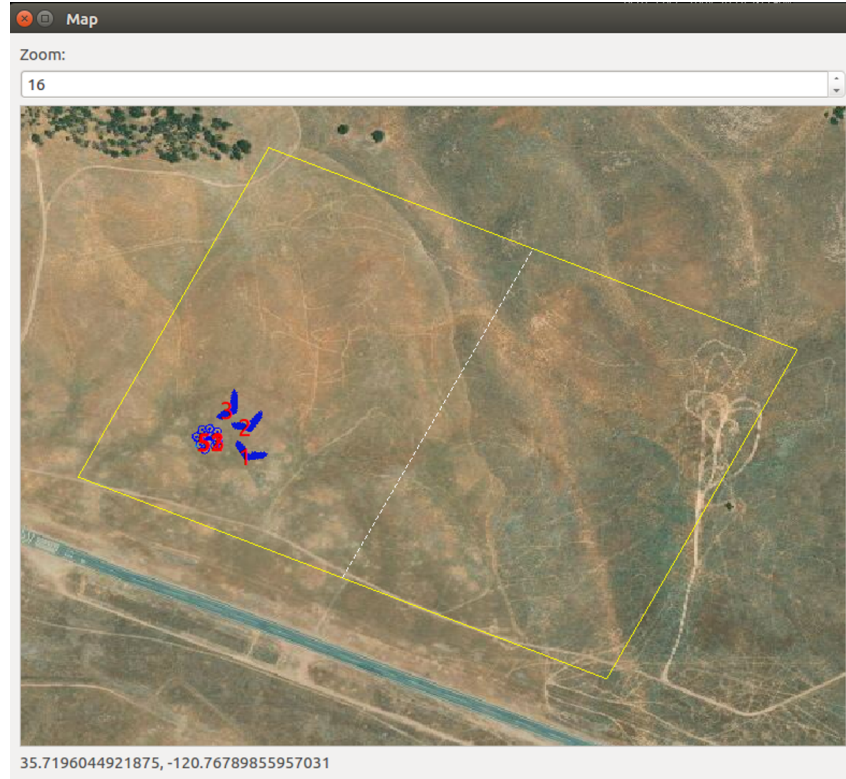


Figure 4.2. The depicted swarm of three quadrotor UAVs (robots 51, 52, and 53) and three fixed-wing UAVs (robots 1, 2, and 3) loiters at a prescribed standby location prior to play commencement. The closely grouped UAVs compute similar task valuations due to their proximity to one another making the initial auction the most competitive and difficult to resolve.

The behavior that is the focus of this thesis is the robot swarm’s response to contacts of interest. We wanted to show that the swarm could effectively exchange information about where contacts were found and seamlessly integrate the new tasks into the existing pool of tasks available for auction. Additionally of interest was the auction algorithm’s computational performance as swarm size was increased in scale. In Section 2.2 we reviewed literature that posited that Bertsekas’ approach to the assignment problem could be shown to be linear in time-complexity if the utility function and epsilon met certain criteria. We used our simulations to empirically demonstrate this phenomenon in the ARSENL robot swarm running our implementation.

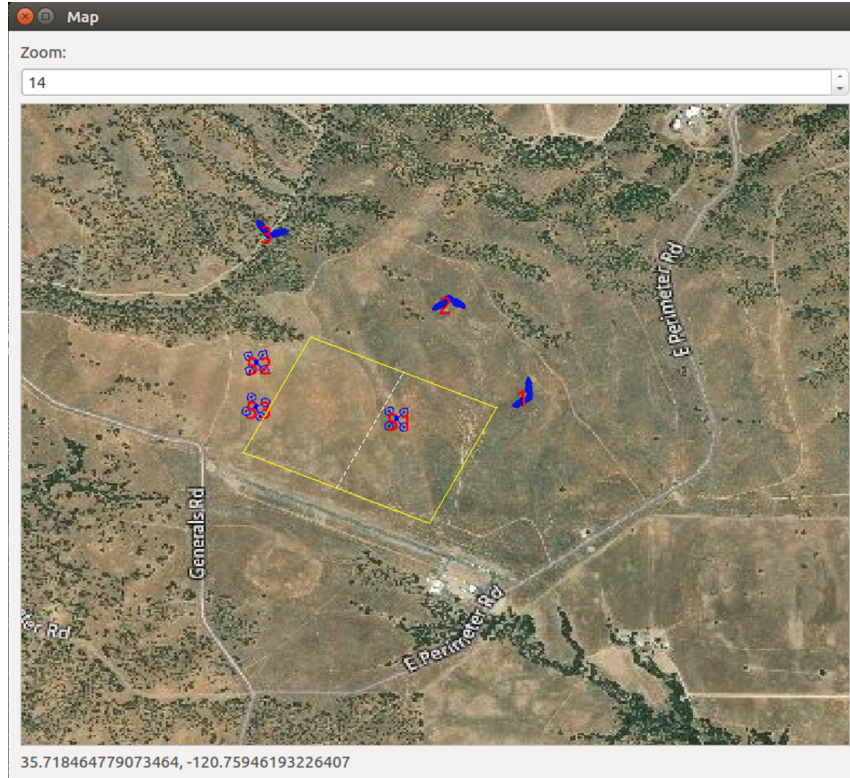


Figure 4.3. The three quadrotor UAVs (robots 51, 52, and 53) in the depicted swarm are significantly slower than their fixed-wing counterparts (robots 1, 2, and 3) so longer transits to more distant cells results in lower utility valuations. Quadrotors therefore submit higher bids for closer cells resulting in the observed desirable cell allocation.

4.2.2 Live Flight Observations

The first live flight experiment with the cued search behavior was conducted in April 2019 and allowed us to fine tune the implementation and identify areas for improvement. The experiment was conducted with a homogeneous swarm of Zephyr II UAVs and included only search tasks. This experiment demonstrated that our implementation roughly replicated the swarm behavior produced in [10]. Our goal was to ensure the UAVs could successfully complete auctions and conduct searches in a predictable manner in various swarm sizes. We observed that the Zephyr II UAVs were successfully utilizing the auction algorithm to assign search tasks and conducting searches as expected with swarms of up to ten robots. Additionally we ran preliminary tests on heterogeneous swarms comprised of Zephyr II fixed-wing UAVs and Mosquito Hawk quadrotor UAVs shown in Figure 4.1. The utility

function used was from [10], however, and only accounted for distance from the robots' locations rather than time required to complete the task. Since the utility function was based on distance, not speed, all UAVs were treated identically (i.e., heterogeneity was not accounted for). The robots closest to the cells were therefore assigned the task regardless of type. Despite this shortcoming and the fact that cueing was not yet implemented, the experiment did verify that the auction algorithm efficiently handled the static tasks.

Once we validated the homogeneous swarm implementation with search tasks, incorporation of heterogeneity of vehicles and tasks into the architecture was fairly straightforward. As described in Section 3.5.1, we implemented a new task object class called `ContactInvestigation` that had the same attributes as the `SearchCell` class for seamless incorporation into the auction. Similarly, the utility function was updated to account for heterogeneous swarm compositions as described in Section 3.6. Utility valuation weights were manipulated based on experimental observations so that robots would bid preferentially to tasks that were best suited to their capabilities. In May 2019, we conducted a second round live-fly experiments to validate the code updates and ensure that the observations we obtained in SITL simulations accurately reflected the behavior of the UAVs in the field.

In total we collected data from eighty simulations with eight different swarm sizes. In this section, we describe how the output was collected and measured, how robots were allocated to tasks based on the type of airframe and the type of task, the overall performance of the swarm search and contact generation, and an analysis of the runtime results. In subsequent sections, we discuss the implications of the findings.

We captured the following performance metrics and aggregated the results across the simulations for each swarm configuration:

1. Auction Time: the amount of time a robot is engaged in the auction before resolving to a task assignment.
2. Idle Time: if the robot's next assignment is dependent on the results of an auction, the robot will idle in place, waiting for the auction to resolve. Robots are capable of participating in an auction and conducting a task at the same time. If the auction resolves before the robot finishes its task, the idle time will be 0.
3. Transit Time: the amount of time a robot is transiting from one location to the start of a task.

4. Search/Investigate Time: the amount of time a robot spends in the actual completion of a task
5. Robot Utilization: the ratio of time spent by a robot searching and investigating relative to the total time it takes the robot swarm to complete the search and associated contact investigations.

A typical output for a swarm comprised of three fixed-wing UAVs and two quadrotor UAVs from one run of the `auction_searcher_test` play is shown in Figure 4.4.

4.3 Data Collection and Analysis

4.3.1 Analysis of Investigation Task Assignment Results

Quadrotors have the advantage of being able to hover in place at a low altitude and are therefore better suited to perform investigation tasks. Since the quadrotor local utility function described in Section 3.6 was designed to prioritize investigation tasks over search tasks, we expected to see quadrotors assigned to most investigation tasks as the experiment progressed. We found, however, that quadrotors did not always win bids for investigation tasks in practice.

The utility function takes into consideration the time it takes for the aircraft to complete its current task, the transit time, the time to complete the task being auctioned, and the airframe type. If the contact is too far away or if the quadrotors are busy with other tasks, the globally preferable assignment for the swarm might be to assign the task to a less well equipped fixed-wing aircraft. Consider Figure 4.3 where the natural tendency of our swarm is to spread away from the start point with the fixed-wing aircraft traveling the farthest. When the more distant fixed-wing UAVs find contacts, the investigation tasks are no longer as profitable to the quadrotors because of the distance to the task. Furthermore, if the swarm is faced with multiple available investigation tasks and fewer quadrotor UAVs, the swarm will correctly allocate fixed-wing UAVs to the investigation tasks instead of passing the task to the next round of auctions because the utility function makes contact investigation more important than continued search to both fixed-wing and quadrotor UAVs.

Figure 4.5 demonstrates how smaller swarms struggle to assign investigation tasks to quadrotors as they become too busy to investigate contacts. As swarm size increases, the swarm

UAV 1 Fixed Wing					
Task Type	Auction Completion Time	Transit Time	Investigate Time	Search Time	Idle Time Prior to Task
SEARCH	2.66	13.60	0.00	50.80	2.60
SEARCH	0.90	3.56	0.00	41.80	0.00
SEARCH	1.30	16.14	0.00	48.20	1.30
SEARCH	0.90	21.73	0.00	48.30	0.90
SEARCH	1.30	13.89	0.00	41.80	0.00
Null	1.70	N/A	N/A	N/A	0.00
UAV 1 Robot Utilization:	0.758				

UAV 2 Fixed Wing					
Task Type	Auction Completion Time	Transit Time	Investigate Time	Search Time	Idle Time Prior to Task
SEARCH	2.65	11.37	0.00	41.30	2.59
SEARCH	0.90	24.13	0.00	49.90	0.00
SEARCH	1.30	4.27	0.00	45.40	0.00
SEARCH	0.90	14.53	0.00	50.20	0.00
SEARCH	1.30	17.45	0.00	41.30	0.00
INVESTIGATE	1.70	9.48	20.00	0.00	0.00
Null	1.70	N/A	N/A	N/A	0.00
UAV 2 Robot Utilization:	0.747				

UAV 3 Fixed Wing					
Task Type	Auction Completion Time	Transit Time	Investigate Time	Search Time	Idle Time Prior to Task
SEARCH	2.61	26.66	0.00	48.70	2.56
SEARCH	0.90	3.56	0.00	41.70	0.00
SEARCH	1.29	50.45	0.00	47.90	0.00
SEARCH	1.30	4.27	0.00	45.60	0.00
Null	1.70	N/A	N/A	N/A	1.70
UAV 3 Robot Utilization:	0.673				

UAV 51 Quadrotor					
Task Type	Auction Completion Time	Transit Time	Investigate Time	Search Time	Idle Time Prior to Task
SEARCH	2.90	5.54	0.00	91.40	2.83
INVESTIGATE	0.90	3.19	20.00	0.00	0.00
SEARCH	1.29	28.00	0.00	85.70	0.00
Null	1.70	N/A	N/A	N/A	0.00
UAV 51 Robot Utilization:	0.833				

UAV 52 Quadrotor					
Task Type	Auction Completion Time	Transit Time	Investigate Time	Search Time	Idle Time Prior to Task
SEARCH	2.65	9.72	0.00	90.02	2.59
INVESTIGATE	0.90	0.07	20.00	0.00	0.00
SEARCH	1.30	6.69	0.00	91.80	0.00
INVESTIGATE	1.30	0.07	20.00	0.00	0.00
Null	1.70	N/A	N/A	N/A	0.00
UAV 52 Robot Utilization:	0.921				

Figure 4.4. The output of three fixed-wing and two quadrotor UAVs in a heterogeneous swarm of five UAVs. Fixed-wing UAV 2 was assigned to an investigate task because the contact location was too distant to be a valuable assignment for any of the quadrotors. As the data indicates, quadrotors are not often assigned to tasks with long transit times.

has more options for assignment and the quadrotor UAVs are more frequently assigned to investigation tasks. For example, in a swarm of three UAVs, the fixed-wing UAVs were likely to win investigation tasks 27 percent of the time; however, in swarms of eight or more UAVs, the fixed-wing UAVs were unable to win more than 3 percent of the investigation tasks across all ten simulations.

We have successfully demonstrated that robot swarms can generate new contact investi-

gation tasks and incorporate them into the auction as we set out to accomplish with our architecture described in Section 3.4.3. Additionally, we observed that the swarm, utilizing the auction algorithm, was able to allocate these new tasks to the closest quadrotor UAV when it was available or the closest fixed-wing UAV in specific situations. These findings indicate that a heterogeneous swarm can use a market-based distributed algorithm to allocate spontaneously generated tasks associated with an area search in an intuitively optimal manner. Moreover, we assert that our results suggest that this approach can be successfully applied to a broad array of dynamic task assignment problems.

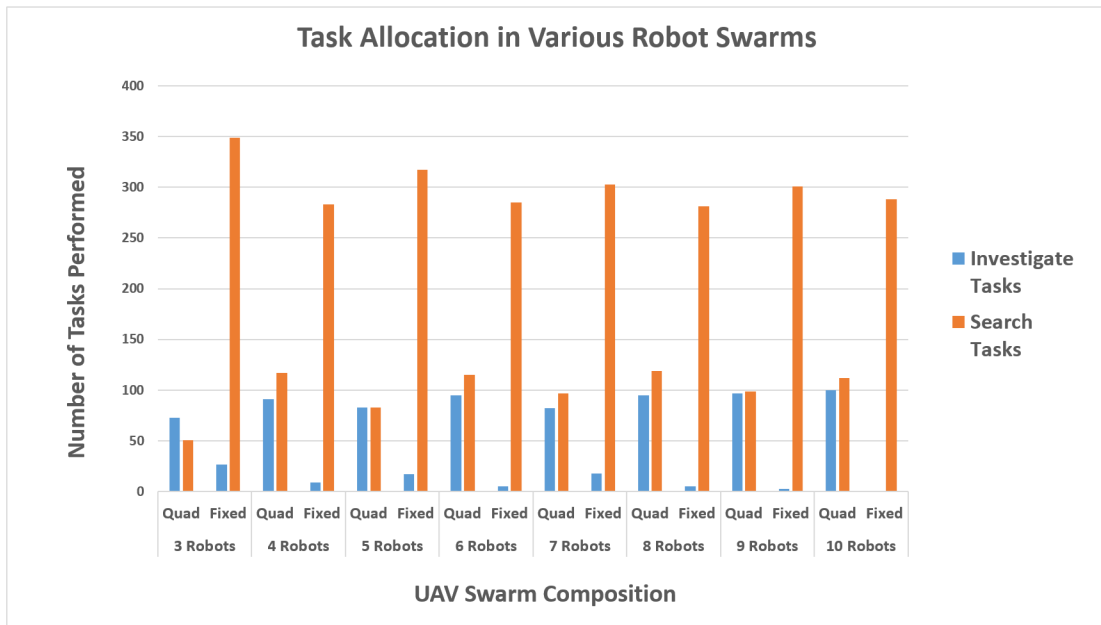


Figure 4.5. The results of ten simulated swarm compositions varying in size from three-ten robots. Odd numbered swarms have one fewer quadrotor UAV than fixed-wing UAVs. As the swarm size increases, quadrotor UAVs are more likely to be assigned to investigation tasks and fixed-wing UAVs are more likely to be assigned to search tasks, resulting in the expected behavior.

Though adding more robots to the swarm can yield a more desirable behavior (i.e., quadrotors winning 100 percent of the investigation tasks), increasing the swarm size has a diminishing effect on the overall performance of the swarm's area coverage. As shown in Figure 4.6, the improvements we saw in the behavior of assigning the best robot to a task did not result in a more efficient coverage of the area. The plateau of the area coverage time is related to decreased convergence time of the auction algorithm and decreased robot

utilization as swarm size increases. In the following Sections, we discuss the extent to which these two factors affected our implementation.

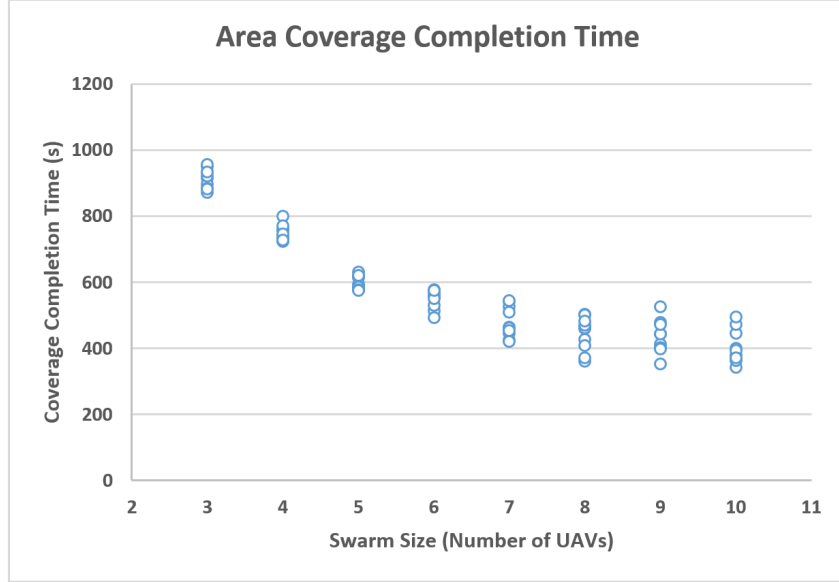


Figure 4.6. An increase in swarm size initially results in a more efficient coverage of the area; however, the benefit diminishes as the environment becomes saturated with robots.

4.3.2 Analysis of the Auction Algorithm

As discussed in Section 2.4 the runtime of the auction algorithm is highly dependent upon the inputs, namely the maximum value of the tasks, the number of tasks, and ϵ . Throughout our simulations, we maintained these three variables as constants and only changed the number and composition of robots in the swarm so that we could evaluate the scalability of our UAS.

In order for the auction algorithm to terminate, all robots must be “happy”. As robots compete for tasks, the tasks’ locally maintained values are constantly changing as the auction progresses until all the robots are “happy” with their assignment. This competitiveness can result in some robots idling and resubmitting the same bid over consecutive rounds until all robots have been assigned to a task. A greater number of auction participants relative to the number of available tasks results in more competitive auctions that increase the median resolution time as demonstrated by Figure 4.7. The pseudo-polynomial nature of the

algorithm becomes apparent when increasing the swarm size as the variance in resolution time also increases in larger swarms. This presence of heteroscedasticity in the auction resolution time makes it difficult to prove the algorithm’s nearly-linear complexity without significantly more data. Nevertheless, we can observe a nearly-linear trend in the mean and median auction resolution times as the swarm size increased, with a linear correlation coefficient of 0.99 and 0.97, respectively (Figure 4.8). We assert that our implementation is tractable on small swarms and given the construct of our implementation could be flexible enough to accommodate a wide variety of tasks (i.e., follow, defend, or attack tasks). We discuss a future research approach to definitively demonstrate the scalability and flexibility of a robot swarm employing our implementation in Section 5.2.

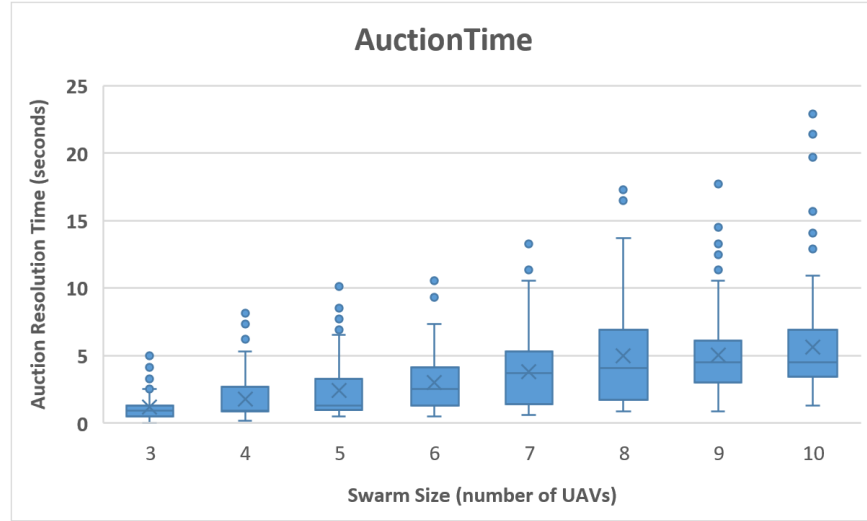
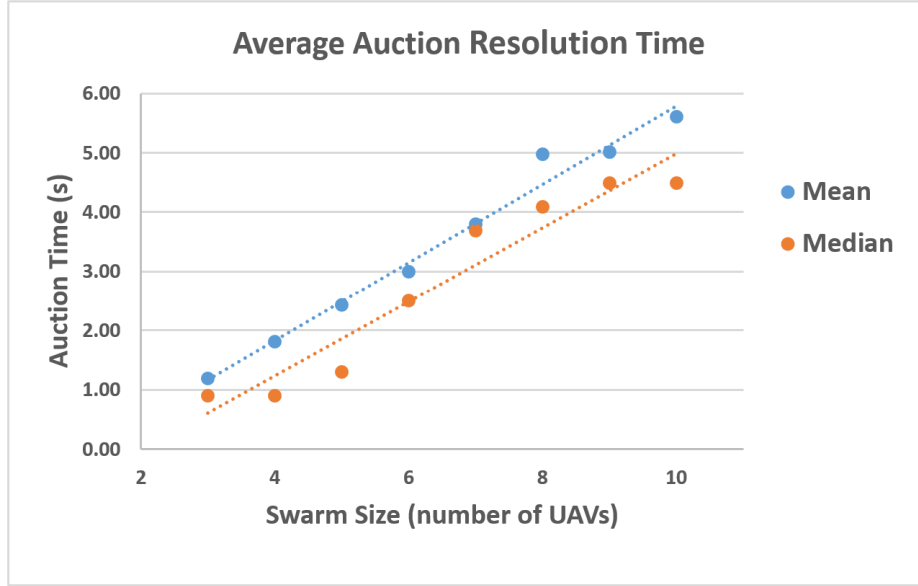


Figure 4.7. As the swarm size increased, auctions were more competitive and took longer to resolve. Multiple robots bidding aggressively for the same task can result in idle periods, as robots who have won their task must resubmit bids over multiple rounds until all assignments have been made.

4.3.3 Analysis of Robot Utilization Results

The robot utilization was measured to determine how well our implementation performed in finding a globally optimal solution using the auction algorithm without unnecessarily delaying the coverage of the search area. We define idle time as the amount of time that a robot is not actively engaged in the execution of either a search task or an investigation task. Our methodology of simultaneously maintaining separate states for the task status and



Average Auction Resolution Time (s) Over Ten Simulations		
Swarm Size	Mean	Median
3	1.18	0.90
4	1.80	0.90
5	2.42	1.30
6	2.99	2.50
7	3.80	3.69
8	4.98	4.09
9	5.02	4.49
10	5.60	4.49
Correlation	0.991	0.965

Figure 4.8. In our experiments with swarms of up to 10 UAVs, the average auction time increased linearly with swarm size. The linear correlation coefficient between swarm size and resolution time is nearly 1.0

the auction status allowed individual robots to simultaneously participate in auctions and execute assigned tasks. The average idle time was thus less than half a second, regardless of the swarm size. Since robots are able to participate in auctions while they are conducting their tasks, they need only initiate a new auction and wait for its results if they do not have assigned tasks in their queues when they complete their current tasks. Since idle time was negligible, the robot utilization is largely a measure of how well the swarm allocated itself to tasks by examining of the transit time in between each of the robots' tasks.

Recall that our idealized robot swarm would achieve a robot utilization of 1.0 across the swarm. For the most part, our swarms achieved robot utilization well above 0.5, indicating that the robot was actively engaged in the search more than half the time. Observed robot utilization results are depicted in Figure 4.9.

The mean robot utilization for fixed-wing UAVs was 0.75. The smallest robot swarm (three UAVs) had the largest mean utilization of 0.81 while the robot swarm with nine UAVs had the lowest mean utilization of 0.69. The opposite trend was observed in the quadrotor UAVs. The mean robot utilization for the quadrotor UAVs was 0.71. The worse mean utilizations were observed in the smaller swarms with a low mean robot utilization of 0.66 in the three-UAV swarm and high mean robot utilizations of 0.76 and 0.74 in the four-UAV and seven-UAV swarm.

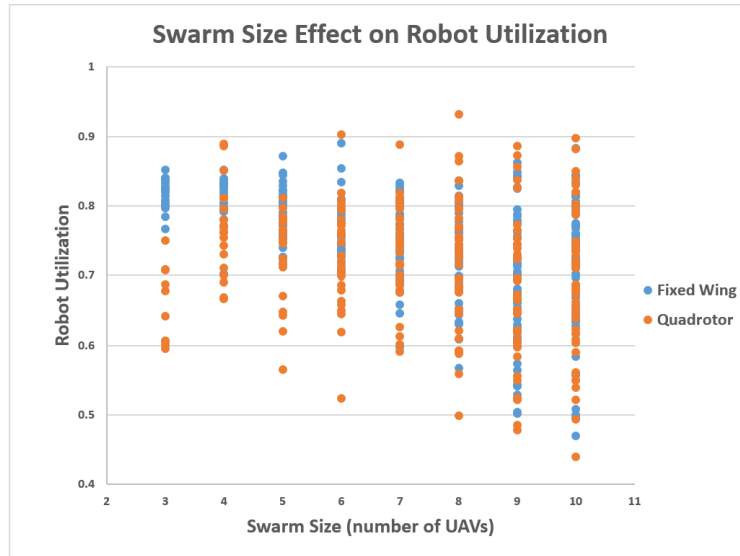


Figure 4.9. Robot utilization for all swarm sizes is largely above 0.5 which indicates that robots are actively engaged in task execution more than half of the time they are executing the play. In smaller swarms quadrotors are overburdened by contact investigations which have high transit times relative to the task execution times lowering their utilization rates. In larger swarms the environment is saturated by the robots leading to longer auction and transit times.

We conjecture that different features of our design and the experiment influence the swarm robot utilization in multiple ways. First, as shown in Section 4.3.1, quadrotors are more

likely to win investigation tasks due to the airframe penalty we imposed in the utility function. The penalty creates market conditions where a quadrotor UAV is partial toward an investigation task even if it is farther away than a search task. As contacts are discovered, the fixed-wing UAVs will continue their search, systematically moving from cell to adjacent cell while a quadrotor UAV is likely to transit long distances to execute assigned investigation tasks. This preferred valuation of investigation tasks over search tasks would explain the slightly lower robot utilization among the quadrotor UAVs across all simulations.

Secondly, the difference between the opposing trends can be explained by a combination of robot saturation and the previously discussed airframe penalty. The robot utilization for fixed-wing UAVs declines as the swarm gets larger because the swarm is saturated with robots. More robots results in longer, more competitive auctions and robots are more likely to win their second, third, or worse choice. These unfavorable assignments are likely associated with longer transit times. On the other hand, with respect to quadrotor UAVs, in smaller swarms the overburdened quadrotor UAVs are more likely to make longer transits to reach farther contact investigations. When the swarm is larger, as the UAVs spread out, there are more options for assignments that do not require such long transits but swarms larger than seven again suffer from the effects of saturation. The overall consistency of favorable robot utilization measurements above 0.5 across multiple swarm sizes indicates that the auction algorithm is an effective method for allocating heterogeneous robots to heterogeneous tasks. Ultimately, developers will have to determine the appropriate number of robots to include in an auction in order for the desired swarm behavior with respect to a particular task.

4.4 Summary

This chapter described the results obtained from experimentation with our cued search implementation in the ARSENL SITL simulation environment. SITL simulation results were validated in two live-fly events conducted at Camp Roberts, California.

Our results demonstrate that the auction algorithm is an appropriate mechanism for autonomously allocating heterogeneous tasks among a robot swarm. Furthermore, we determined through experimentation that our implementation of the auction algorithm was appropriate for swarms of up to 10 UAVs and likely scalable to much larger swarms. We

were also able to convey that increasing size and heterogeneity of the swarm does not degrade the performance of the auction algorithm.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:

Conclusion

In his NPS master’s thesis [10], Matthew Hopchak demonstrated that multiple robots in coordination could operate autonomously using the auction algorithm; a method for solving the assignment problem proposed by Bertsekas [13], to assign tasks to individual robots in a large area search scenario. Hopchak’s research was applied to heterogeneous robots that are required to accomplish homogeneous tasks (i.e., dissimilar fixed-wing UAVs searching small cells within the search area).

In this thesis, we extended Hopchak’s work by increasing swarm heterogeneity to include quadrotor UAVs and task heterogeneity to include contact investigation tasks. In addition, we demonstrated the ability to dynamically introduce follow-on tasks to the auction as investigation contacts are identified.

5.1 Framework Analysis and Areas for Improvement

In Section 1.1, we discussed a rubric for evaluating swarm technology as flexible, scalable, and robust. The focus of our implementation was developing a swarm that was inherently all three. Flexibility was achieved by leveraging an object orientated design approach. The auction algorithm facilitates scalability if the input values are carefully defined to achieve pseudo-linear growth as swarm size increases. Our current implementation has at least one limitation in robustness. Specifically, the auction process will fail to terminate if confronted with robot failures. We propose a simple modification to the implementation, however, that can alleviate the problem.

5.1.1 Flexibility

The utility function is the core of the decision making logic for the robots. Not surprisingly, changing the utility function can result in vastly different behaviors for robots within the swarm. Our utility function is relatively simple: robots value tasks based on the amount of time it will take them to complete the task and the suitability of the task to their capabilities. There are other metrics that are plausible for a utility function, and the suitability of these

should be formally assessed. For example, task prioritization could be factored into task value more deliberately. For instance, our cued search system might be extended to include an associated level of certainty for detected contacts. Investigation of contacts with high (or alternatively low) levels of certainty might reasonably take precedence. Making small changes to the utility function might provide a simple way to increase the swarm’s apparent “intelligence” and merits further exploration.

There are currently two task templates in our implementation: investigation and search tasks. These parent tasks lay the foundation for other similar tasks. Leveraging our object oriented design’s inheritance feature, any future work can easily incorporate new tasks by deriving new task objects with the methods and attributes that are already defined in the existing framework. Identification of additional mission types and tasks that swarm UAVs might execute in their accomplishment is warranted.

Finally, addressing the dynamicity of swarm and task composition in the context of the assignment problem is a potentially important issue. For example, one possible application of swarm robotics is persistent surveillance. The basis for persistent surveillance is in the area coverage problem that we solve with our implementation; however, there are added challenges to maintaining coverage continuously over an arbitrary period of time. In particular, the re-incorporation of search cell tasks into the auction in a way that minimizes the amount of time between revisitations must be addressed. Equally important is the process by which new UAVs are introduced to the UAS to relieve ones whose endurance has been exhausted.

5.1.2 Scalability

Our experiment assessed the performance of swarms of up to 10 UAVs in two search areas: a large area of 30 cells and a smaller area of 12 cells. In both experiments, the swarm worked equally well with relatively small swarms of three robots as it did with larger swarms of ten robots by leveraging the distributed nature of the auction algorithm. The effect of increasing swarm or task set size appears to have a negligible effect on performance. This makes intuitive sense since the auction algorithm is soundly based on a decentralized computing approach and exhibits pseudo-linear computational growth if certain criteria are met. This phenomenon has not, however, been empirically verified for our system.

Based on our experiments, the algorithm scales well when the swarm size remains small in proportion to the number of tasks (e.g., swarm size of three robots searching 30 cells scales well to six robots searching 30 cells); however, the auction takes significantly longer to terminate as the environment approaches UAV saturation. Consider, for example, 50 UAVs competing for assignment to 12 tasks. Many of the robots will be competing for the same assignments, so the complete auction can require many rounds.

To a certain extent, discretizing a search (or any complex task) into more subtasks might promote scalability, but it presents other problems. The first problem is that discretizing too much may be unreasonable (e.g., a search task with one waypoint). Additionally, discretizing itself adds computational overhead and presents its own challenges. Other approaches to the issue of robot-to-task saturation should therefore be explored.

Another aspect of scaling that our implementation does not address well is Erol Sahin’s notion of “scaling in time” [4]. Similar to the swarm composition issue discussed in the previous section, scaling in time is the ability of the swarm to grow or shrink in size during play execution. Our system does not currently scale in time; however, the foundation exists for its incorporation. As resources are added to the auction (i.e., through the process of cueing in our implementation), the swarm might autonomously decide how to scale itself to optimally meet the task demands by limiting the number of robots participating in the swarm. Conversely, if it is determined that the auction resources are not being assigned efficiently enough, more robots can be allocated to the task.

A final scalability issue that might be addressed is the selection of auction algorithm ϵ values. The value of ϵ plays a significant role in both the convergence of the algorithm and the quality of the solution. A small ϵ will result in highly competitive bidding where robot bids increase in small increments. This can require many rounds to reconcile winners. A large ϵ , on the other hand, converges more quickly but can result in suboptimal assignments. Bertsekas has shown that the solution is globally optimal within $n\epsilon$ [13]. The larger ϵ , the more likely it is for robots to be assigned suboptimally.

Bertsekas offers a modification to the auction algorithm that solves the problem of run time convergence and accuracy called ϵ -scaling [13]. The idea is that ϵ values vary over the course of the auction starting with a large values to encourage rapid convergence and progressing to smaller values later to improve the optimality. This can drastically reduce the number of

rounds overall by increasing prices rapidly at first. Smaller ϵ values later more refine earlier suboptimal results to more accurately match agents with higher utility assignments. While our implementation performed well in our experiments, this approach is worth exploring to improve the scalability of the ARSENL auction algorithm implementation to larger and more heterogeneous swarms.

5.1.3 Robustness

Robustness is not explicitly built into the ARSENL swarm UAS; however, the distributed nature of the auction algorithms facilitate a more robust implementation. One vulnerability of the current implementation is that during auction execution, the synchronization step requires receipt of bids from all the participants. Failure detection is not currently incorporated, so there is no existing mechanism for the robot to determine when another participant is malfunctioning. Therefore, if any robots fail, the auction will freeze and fail to converge. The ARSENL swarm does, however, incorporate failure detection more generally [30]. Incorporation and testing of this capability in the context of auction-reliant behaviors can greatly improve the swarm's resilience in the face of robot failures.

Additional approaches to preventing a global swarm failure due to a single robot's failure might also be implemented in a fairly straightforward manner to provide additional robustness. A crude approach, for instance, might utilize timed synchronization. That is, if a bid is not received from a robot within a predetermined amount of time, the swarm can conclude that the robot no longer participating and proceed accordingly.

5.2 Additional Future Work

Among the more general ways in which the work of this thesis might be extended is with greater heterogeneity among UAVs, tasks, and even across different operational domains (e.g., aerial and terrestrial). Incorporating more information-dense sensor data into utility calculations can provide opportunities to increase the swarm's apparent intelligence. Lastly, an exploration of combinatoric auctions as a means of obtaining more optimal solutions might prove beneficial.

5.2.1 Heterogeneity

We examined two aspects of research: robot heterogeneity and task heterogeneity. To represent the heterogeneity of robots we used fixed-wing UAVs that could quickly cover large distances in a short amount of time and quadrotor UAVs that could hover in position at a waypoint but were slow moving with limited endurance. Increasing heterogeneity is partially a scalability problem but, it is different enough that it has unique challenges as well. We have shown that market-based algorithms can produce near-optimal assignments for multiple platforms with different capabilities to multiple tasks with different requirements. There are, however, a number of approaches to further experimentation with auction algorithms and heterogeneity. The following list provides a short description of a few of these potential areas of inquiry:

1. **Increased task complexity.** Include the same robots with the same capabilities as the UAVs in this thesis, but increase the complexity and number of possible tasks. This might involve developing more complex approaches to search cell coverage, incorporation of minimum probability of detection or confidence in detection into utility calculations, development of more dynamic investigation tasks (e.g., varying loiter time based on vehicle or contact type), etc. Since task values and utility costs are currently calculated as a function of time, many of these additions can be added to the current implementation without significant changes to the methodology. Increased task complexity facilitates the increased flexibility of the swarm as discussed in Section 5.1.1.
2. **Ordinal tasks.** Apply the market-based approach to different swarm objectives. The global objective that we defined for the swarm was divided into loosely coordinated independent tasks. The desired behavior was a systematic organized search but there was no requirement that the robots complete subtasks in any specific order. A more complex global objective might include highly dependent tasks that must be completed in a particular order (e.g., search involving active and passive sensors where cells must be searched with the passive sensor first).
3. **Increased capability heterogeneity.** Our swarm consisted of robots with two capabilities: rotary-wing and fixed-wing aerial search. As more complex tasks are added to the swarm, robots will require additional capabilities (e.g., different sensor packages on otherwise similar platforms). A potential research problem would be to

examine how a diverse set of capabilities and limitations can be leveraged using a market-based approach. Section 5.2.2, describes how these added capabilities would necessitate a more complex utility function.

4. **Multi-domain robot swarming.** Examine the applicability of the auction algorithm to robot swarms that cross domains. Douglas Horner, for instance, explores the possibility of multi-vehicle collaboration between UAVs and UUVs in [31]. In his UAS, some robots would not be capable of accomplishing tasks that others would if the task did not belong to their domain (e.g., a UAV is incapable of performing an underwater search). A market-based distributed algorithm similar to our auction algorithm implementation might address many of the issues that he is concerned with in a more robust or flexible manner.

One final potential research topic focused on swarm heterogeneity would be study of heterogeneity quantification as more vehicle capabilities are integrated into the system. That is, addressing the question of rigorously capturing both the level of heterogeneity in the system and also the effectiveness of our implementation. This would effectively extend the work of Balch in [6] by applying his techniques to the ARSENAL swarm UAS.

5.2.2 Complex Valuation Methods

In Section 5.1.1, we briefly discussed modification of the utility function to adapt to different types of problems. In this section, we discuss specific options for incorporating more advanced information into the utility function.

Detection events utilized to spur investigation tasks were artificially introduced into the system for our research. These events, though suitable for our purpose, were not realistic analogs of detection events generated over the course of an actual search. One area of research would be to evaluate different onboard sensors' ability to generate detection events. For example, a Magnetic Anomaly Detection (MAD) device provides a different output than a camera. The information that can be gleaned from different sensors will be varied. Even if events generated by both sensors spawn the same follow-on task, the data source might affect how suited a particular vehicle is to conducting the follow-on investigation (e.g., an infrared-equipped UAV might be more suited to investigating the camera-sourced contact).

These and other sensors can supply information to a utility function to help the robots

make more informed decisions about how to value tasks using our implemented auction algorithm. Research into accounting for their capabilities in both task valuation and cost functions is required to leverage this information.

Another limitation of our utility function is that it does not use the disposition of the rest of the robot swarm as a factor for valuation. Each robot values tasks independently of what the rest of the swarm is doing. As we saw in our implementation, there would be scenarios where threats are found and the UAVs best suited to those tasks are too far to make winning bids on the task. One approach that future research might explore is the development of a utility function that could devalue tasks that place the robot swarm into disadvantageous dispositions. For example, a penalty can be imposed for clustering too many robots of the same type in one area or for leaving a high-value unit vulnerable.

The utility function drives the autonomy of the swarm's allocation of robots to tasks. An expansion of heterogeneity to the swarm will be associated with a more complex utility function that defines how robots will bid for tasks. By establishing a basis for a robot's task preference, a sensible utility function can be developed and tested in the ARSENL SITL simulation environment as we have done in this thesis.

5.2.3 Designated Subswarms

Our implementation conducts auctions in which all swarm UAVs participate making it possible for robots to be paired with tasks that are not optimal. Furthermore, a robot that is burdened by a suboptimal task may be repositioned to a location that is disadvantageous to the swarm. Consider a swarm of robots that is tasked to provide defensive surveillance (i.e., patrol tasks analogous to search and blocking tasks analogous to contact investigation). In our implementation quadrotor UAVs can be allocated to patrol tasks that are far from the defended position if no blocking tasks are available. This might place the quadrotor out of position to be assigned emergent blocking tasks to which it is best suited.

One approach to addressing this issue might be to divide the swarm into two or more subswarms. The initial task assignment for the surveillance example would be accomplished by one subswarm leaving the other in reserve (i.e., available for immediate assignment to emergent tasks). The subswarm divisions would be based on the mission requirements and the types of tasks each subswarm was expected to accomplish. Instead of finding tasks

and entering them into the existing auction as we have done, new tasks can be passed to the appropriate subswarm for bidding based on the subswarm's composition (i.e., to the subswarm most suited to the task). This research would likely involve exploration of various subswarm definition approaches. An additional research initiative associated with the same thread might involve the evaluation of the extent to which a subswarm-based system would conserve (or potentially waste) resources over the course of an entire mission. For example, since the system could potentially guarantee that it would not have to utilize quadrotor UAV resources to accomplish search tasks for which they are poorly suited, investigation tasks to which they are well suited might be more efficiently accomplished. On the other hand, if there are no tasks for a particular subswarm to accomplish (e.g., no investigation tasks are spawned), then the subswarm's resources are essentially wasted. A rigorous study of the likely outcomes of different subswarm configurations in various scenarios is therefore warranted.

5.2.4 Combinatoric Auctions

Use of a combinatoric auction to solve the assignment problem is discussed in Section 2.4.4 and identified in [10] as a potential area of follow-on research. Although a combinatoric auction can potentially derive an optimal solution, the number of potential options makes its full implementation intractable. In order to produce a feasible implementation, therefore, enough constraints must be imposed to mathematically preclude a combinatoric explosion. The constraints would likely set the maximum number of tasks to which any agent could and they would be a function of the number of tasks and the number of agents participating in the auction. Computational complexity notwithstanding, the potential optimality of the solutions and requirement to assign a large number of tasks via a single auction make combinatoric auctions a potentially attractive mechanism for task assignment within the swarm.

5.3 Summary and Conclusions

Our research demonstrates that a heterogeneous robot swarm can autonomously allocate UAVs to heterogeneous tasks using a distributed market-based algorithm. As autonomous robot swarms become more integrated into the military's missions, increased heterogeneity will enable their application to a broader range of military operations. The flexibility

of the UAS presented in this thesis establishes a foundation for future research to develop implementations that employ specific heterogeneous capabilities adapted to solve challenges that are not feasible for current homogeneous systems.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: Source Code

A.1 auction_searcher_test Source Code

```
#  
# Dr. Duane Davis 2019  
# LT Britt Campbell 2019  
#  
# Standard python imports  
import math  
import time  
import os  
import re  
import datetime  
# ROS imports  
import rospy  
import random  
  
# ARSENL imports  
import arsenl_control.swarm_control as ctrl  
import arsenl_control.plugin_behavior as plugin  
import arsenl_behavior_tools.auctions as auctions  
import arsenl_behavior_tools.searcher as search  
import arsenl_behavior_tools.waypoint_paths as paths  
import arsenl_data.enumerations as enums  
import arsenl_algorithms.independent_transits as transits  
#bc: utility imports  
import arsenl_library.math_utils as ro_math  
import arsenl_library.gps_utils as gps  
import arsenl_library.bitmapped_bytes as bytes
```

```

class AuctionCue(plugin.PluginBehavior):
    ''' Simple test behavior for SingleItemAuction
        functionality

    Class member variables:
        _resources: tuple of dummy resources for the auction
        _auction: single item auction object
        _driver: direct driver to orbit the standby location

    Inherited member variables (PluginBehavior):
        id: Unique integer identifier for this behavior
        _controller: SwarmController object to which this
            behavior belongs
        _participants: Iterable object of behavior-
            participating vehicle IDs
        __ctr: used to pause between auctions

    Class member functions:
        parameterize: implementation of the PluginBehavior
            virtual function
        compute_command: runs one iteration of the behavior's
            control loop
        process_behavior_data: handles behavior-specific
            communications

    Inherited member functions (PluginBehavior)
        set_controller: sets the controller object for this
            object
        set_participants: sets the list of participating
            vehicles
        get_name: returns the string name of the behavior
        set_ready: safely sets the ready state to True or

```

```

        False
    is_ready: returns the behavior's current readiness
        state
    set_active: safely activates or deactivates the
        behavior
    is_active: returns the behavior's current active state
    activation_time: returns time at which the behavior
        was activated
    time_since_activation: returns elapsed time since
        activation
    get_status: returns the behavior's current ready and
        active status
    , , ,

```

```

RESTART = 0
IN_PROGRESS = 1
INACTIVE = 2
COMPLETE = 3

```

```

SEARCH = 4
SEARCH_WAIT = 5
INVESTIGATE = 6
IDLE = 7

```

```

# Class-specific enumerations and constants
# Basic rectangular search area enumerations (no
    obstacles)
BASIC_LIVE_FLY = 10
BASIC_LARGER    = 11
# Complex polygonal search area enumeration (with
    obstacles)
COMPLEX = 12

```

```

# Search area southwest location
AREA_SW_LAT = 35.721147 # these values will be
    modified when generate search area is called
AREA_SW_LON = -120.773008

# other enumerations
AREA_MIN_ALT = 354
AREA_MAX_ALT = 854
NUM_CELLS = 12 # this is modified by the cell
    generation methods below
NOT_BIDDING = NUM_CELLS

def __init__(self, behavior_id, behavior_name,
    controller):
    ''' Class initializer initializes class variables.
    @param behavior_id: unique identifier for this
        behavior
    @param behavior_name: string name of this behavior
    @param controller: Controller object to which this
        behavior belongs
    '''

    super(AuctionCue, self).__init__(behavior_id,
        behavior_name, controller)
    self._resources = []
    self._auction = None
    self._driver = None
    self._wp_driver = None
    self.__ctr = 0
    self._auction_iteration = 0
    self.__auction_state = AuctionCue.RESTART
    self.__task_state = AuctionCue.SEARCH_WAIT

```

```

self.current_assignment = None
self.next_assignment = None
self.goto = (0,0)
self.wpt_ctr = 0
self.compute_cmd_ctr = 0
self.investigation_id = 0
self.inside_search_box = False
self.time_start = 0.0
self.commence_investigate = False
self.time_till_investigate = 0.0
self.detection_occured = False
self.cell_detection_list = []
self.investigation_time = 20.0
self.auction_idle_time = 0.0
self.qr_mult = 5.0
self.fw_mult = 5.0

#self._choose_search_area = AuctionCue.
    BASIC_LIVE_FLY
self._choose_search_area = AuctionCue.BASIC_LARGER
self._follow_wp = True

#Data collection Variables
self.write_to_file = True
self.search_tasks_completed = []
self.investigate_tasks_completed = []
self.idle_start = time.time()
self.time_idle_prior = 0.0
self.search_time_start = 0.0
self.auction_idle_time = 0.0
self.dropped_tasks = 0.0
self.data_compile = []
self.final_auc_time = 0.0

```

```

self.save_path = '/home/arsenl/thesis_data'

def parameterize(self, params):
    ''' Sets behavior parameters based on set service
        parameters
        This behavior requires no parameters (if it's red
        and flies, it dies)
        @param params: parameters from the set service
        request
        @return True if set with valid parameters
        , , ,

#     BASIC_LIVE_FLY: live-fly area (Camp Roberts
#     McMillan Airfield geo-fence safe)
#     BASIC_LARGER:   large-basic area, uniform
#     rectangular cells
#     COMPLEX:        large-complex area, polygonal
#     environ with obstacles and irregular cell sizes
# ----- HARD CODE SELECT ONE OF THE BELOW OPTIONS
# -----

#self._choose_search_area = AuctionCue.
#     BASIC_LIVE_FLY
self._choose_search_area = AuctionCue.BASIC_LARGER

self._resources = []
self._auction = None
self._driver = None
self._wp_driver = None
self.__ctr = 0
self._follow_wp = True

```

```

self._auction_iteration = 0

self.__auction_state = AuctionCue.RESTART
self.__task_state = AuctionCue.SEARCH_WAIT

self.current_assignment = None
self.next_assignment = None
self.goto = (0,0)
self.wpt_ctr = 0
self.investigation_id = 0
self.inside_search_box = False
self.time_start = 0.0
self.commence_investigate = False
self.time_till_investigate = 0.0
self.detection_occurred = False

self.investigation_time = 20.0
self.qr_mult = 2.0
self.fw_mult = 2.0

#Data collection variables
self.write_to_file = True
self.search_tasks_completed = []
self.investigate_tasks_completed = []
self.idle_start = time.time()
self.time_idle_prior = 0.0
self.search_time_start = 0.0
self.auction_idle_time = 0.0
self.dropped_tasks = 0.0
self.data_compile = []

self.final_auc_time = 0.0

```

```

self.save_path = '/home/arsenl'

#Aircraft/Airframe
self.aircraft_type = rospy.get_param("aircraft_type"
    )
# Enumeration for aircraft types
#AC_UNKNOWN = 0
#AC_FIXED_WING = 1
#AC_COPTER = 2
self.airframe_type = rospy.get_param("airframe_type"
    )
# Enumeration for different airframes
#AIRFRAME_UNKNOWN = 0
#ZEPHYRII = 1
#ZEPHYRII_GEN3 = 2
#PENGUIN = 3
#MOSQUITO_HAWK = 4

self._controller.log_info(str(self.aircraft_type))
self._controller.log_info(str(self.airframe_type))
ac_id = 'UAV_' + str(self._controller.own_id)
if self.aircraft_type == enums.AC_COPTER:
    ac_type = 'Quadcopter'

elif self.aircraft_type == enums.AC_FIXED_WING:
    ac_type = 'Fixed_Wing'

else :
    ac_type = 'Unknown_Aircraft_Type'

self.data_compile.append(ac_id + ',' + ac_type)
self.data_compile.append('Task_Type,' + Task_ID,

```

```

        Auction_Completion_Time, Transit_Time,
        Investigate_Time, Search_Time, Idle_Time_Prior_to
        Task')
self._controller.log_info("\n*\n**\n" + "I_AM_A_"
    + ac_type + "_\n**\n*")

if self._choose_search_area == AuctionCue.
    BASIC_LIVE_FLY:
    self._choose_search_area = 0

elif self._choose_search_area == AuctionCue.
    BASIC_LARGER:
    self._choose_search_area = 1

elif self._choose_search_area == AuctionCue.
    BASIC_COMPLEX:
    self._choose_search_area = 2

else:
    self._controller.log_info("Invalid_Search_Area")

self.cell_partition = search.SearchAreaPartition(
    self._choose_search_area, self._controller)
self.cell_partition.generateBasicSearchArea()

self._resources = []
self._resource_tracker = []
for cell in self.cell_partition._cells:

    self._controller.log_info("Cell_task_value_is_"
        + str(self.cell_partition._cells[cell].
            _task_value))

```

```

        self._resources.append((cell, self.
            cell_partition._cells[cell]))
        self._resource_tracker.append("c")

num_resources = len(self._resources)
self._controller.log_info("\n*\n**\n" + str(
    num_resources) + "\_Search\_resources\_created.\_
    Search\_Area\_Ready\_n**\n*")

num_detects = int(0.25*num_resources)

spacer = int(num_resources/num_detects)

self.cell_detection_list = []
for i in range(num_resources):
    if i % spacer == 0:
        self.cell_detection_list.append(i)

self._controller.log_info("\n*\n**\n" + str(
    num_detects) + "\_Detection\_events\_scattered\_
    throughout\_search\_area.\_n**\n*")

self._controller.log_info("\n*\n**\n" + str(len(
    self.cell_detection_list)) + "\_this\_should\_be\_the\_
    same\_number\_or\_close.\_n**\n*")

self._driver = transits.DirectDriver(self.
    _controller, \
                                     self.
                                     _controller.
                                     ap_wpt.x, \
                                     self.

```

```

        _controller .
        ap_wpt.y, \
self .
        _controller .
        ap_wpt.z, \
self .
        _controller .
        ac_limits .
        nom_fwd_speed
    )

self._investigate_driver = transits.DirectDriver(
    self._controller, \
        self .
        _controller .
        ap_wpt.x, \
self .
        _controller .
        ap_wpt.y, \
self .
        _controller .
        ap_wpt.z, \
self .
        _controller .
        ac_limits .
        nom_fwd_speed
    )

self._wp_driver = transits.WaypointSequenceDriver(
    self._controller)
startup_sequence = paths.WaypointSequencer()

standby_point_lat = self._controller.last_wp.x

```

```

standby_point_lon = self._controller.last_wp.y

startup_sequence.append(standby_point_lat,
    standby_point_lon, self._controller.ap_wpt.z, \
        self._controller.
            ac_limits.
            nom_fwd_speed)

self._wp_driver.reset(startup_sequence)

self._auction = \
    auctions.SingleItemAuction(self._controller.
        own_id, self._controller)

self._auction.set_local_utility_function(self.
    __distance_utility)

if self._auction.initialize_auction(20.0, self.
    _resources, self._participants):
    self.__auction_state = AuctionCue.IN_PROGRESS
    self._controller.log_info("Commencing initial \
        auction")
    self.auction_start_time = time.time()

else:
    self.__auction_state = AuctionCue.COMPLETE
    self._controller.log_info("No \resources \
        available \for \bid")
return True

def __distance_utility(self, cell_record):

```

```

# transit time + remaining time on station
if self.current_assignment != None and not self.
current_assignment.is_null:
    #if the resource record is an investigate task
    and if you are a copter, do not count your
    time remaining because you will drop task if
    you win
    if self.aircraft_type == enums.AC_COPTER:
        own_posit = self._controller.get_own_state()
            .state.pose.pose.position.lat, \
                self._controller.get_own_state()
                    .state.pose.pose.position.lon
        if self.current_assignment.resource._task_id
            == search.SEARCH_CELL:
            time_remaining = 0.0
        elif self.current_assignment.resource.
            _task_id == search.INVESTIGATE_PT:
            time_remaining = self.current_assignment
                .resource.getTimeRemaining()

    elif self.aircraft_type == enums.AC_FIXED_WING:
        time_remaining = self.current_assignment.
            resource.getTimeRemaining()
        own_posit = self.current_assignment.resource
            .lastWaypoint[0], \
                self.current_assignment.resource
                    .lastWaypoint[1]

else:
    time_remaining = 0.0
    own_posit = self._controller.get_own_state().
        state.pose.pose.position.lat, \

```

```

        self._controller.get_own_state().
            state.pose.pose.position.lon

#speed formula  $v = d/t$ 
# $t = d/v$ 
time_to_transit = cell_record.resource.
    getTaskTransitTime(own_posit)

time to complete task in question
time_to_complete = cell_record.resource.
    getTimeToAccomplishTask(own_posit)

#total time
total_utility_time = time_remaining +
    time_to_transit + time_to_complete

if self.aircraft_type == enums.AC_COPTER:
    if cell_record.resource._task_id == search.
        SEARCH_CELL:
            result = cell_record.resource._task_value -
                self.qr_mult*total_utility_time

    elif cell_record.resource._task_id == search.
        INVESTIGATE_PT:
            result = cell_record.resource._task_value -
                total_utility_time

elif self.aircraft_type == enums.AC_FIXED_WING:
    if cell_record.resource._task_id == search.
        SEARCH_CELL:
            result = cell_record.resource._task_value -

```

```

        total_utility_time

    elif cell_record.resource._task_id == search.
        INVESTIGATE_PT:
        result = cell_record.resource._task_value -
            self.fw_mult*total_utility_time

    else:
        result = cell_record.resource._task_value -
            total_utility_time

    return round(result,1)

def aircraft_idle(self):
    center_field = gps.gps_midpoint(self.cell_partition.
        area_nw_lat, \
            self.cell_partition.area_nw_lon, \
            self.cell_partition.area_se_lat, \
            self.cell_partition.area_se_lon)

    standby_point_lat = self._controller.last_wp.x
    standby_point_lon = self._controller.last_wp.y

    self._driver = transits.DirectDriver(self.
        _controller, \
            standby_point_lat, \
            standby_point_lon, \
            self._controller.ap_wpt.
                .z, \
            self._controller.
                ac_limits.
                nom_fwd_speed)

```

```

def compute_command(self):
    ''' Executes one iteration of the behavior (returns
        a speed
        waypoint associated with the previously determined
        random point)
    '''

    self._auction.send_bid_info()
    if self.__auction_state == AuctionCue.RESTART:
        ready = self._auction.
            continue_auction_from_current()
        if ready:
            self.__auction_state = AuctionCue.
                IN_PROGRESS
            self._auction_iteration += 1
            self._controller.log_info("Commencing next
                auction iteration number: " + str(self.
                    _auction_iteration))
            self.auction_start_time = time.time()
        else:
            self.__auction_state = AuctionCue.COMPLETE
            self.__task_state = AuctionCue.IDLE
            self._controller.log_info("All auctions
                complete")
            self.__ctr = 0

    # when you initialize the auction if there are
    # resources to be bid on, the state will be
    IN_PROGRESS

```

```

elif self.__auction_state == AuctionCue.IN_PROGRESS:
    #auction_step will be false until the round of
    auctions is finished and the bots are
    assigned to tasks

    if self._auction.auction_step():
        self.__auction_state = AuctionCue.INACTIVE
        self._controller.log_info("Auction_State_set
            _to_Inactive")
        self._controller.log_info("round_" + str(
            self._auction_iteration) + "_finished.")
        self.next_assignment = self._auction.
            _assignments[self._controller.own_id]
        #time taken to win task in an auction
        auction_time = time.time() - self.
            auction_start_time

        if not self.next_assignment.is_null:
            self.next_assignment.resource.
                time_to_win = auction_time
        else:
            self.final_auc_time = str(round(
                auction_time,2))

        if self.current_assignment == None:
            self._start_next_task()

if self.__task_state == AuctionCue.SEARCH:

    if self.__ctr == 0:

        if int(self.current_assignment.id) in self.
            cell_detection_list:

```

```

        self._controller.log_info("\n*\n**\n\
            nInvestigation---->Bot" + str(self
                ._controller.own_id) + " is searching
                cell" + str(self.current_assignment
                    .id) + "\n**\n*")
        self._controller.log_info("\n*\n**\n\
            nInvestigation---->Bot" + str(self
                ._controller.own_id) + " is searching
                cell" + str(self.current_assignment
                    .id) + "\n**\n*")
        self._controller.log_info("\n*\n**\n\
            nInvestigation---->Bot" + str(self
                ._controller.own_id) + " is searching
                cell" + str(self.current_assignment
                    .id) + "\n**\n*")

    else:
        self._controller.log_info("\n*\n**\n\
            nBot" + str(self._controller.own_id)
            + " is searching cell" + str(self.
                current_assignment.id) + "\n**\n*")

    if self.__ctr == 15:
        self.notify_task_status(auctions.
            ResourceRecord.ACTIVE)
        self._auction.set_resource_status(self.
            current_assignment.id, auctions.
            ResourceRecord.ACTIVE)

    #if self.aircraft_type == enums.AC_COPTER:
    #    self.drop_task()

    if self.current_assignment.id in self.

```

```

cell_detection_list:
    if not self.detection_occurred and self.
        _wp_driver._sequence.percent_complete()
        > 35:

        self.detection_occurred = True
        self._controller.log_info("\n*\n**\n
            ---!!!DETECTION_EVENT: INVESTIGATE
            !!!---\n**\n*")
        current_pos = self._controller.
            get_own_state().state.pose.pose.
            position.lat, \
                self._controller.
                    get_own_state().state.
                        pose.pose.position.lon
        self.investigate_alert(current_pos)

if self.__ctr % 100 == 0:

    completion_percent = self._wp_driver.
        _sequence.percent_complete()
    self._controller.log_info("Bot" + str(self.
        _controller.own_id) + "\task\completion\
        percentage:" + str(completion_percent))

if not self.inside_search_box:
    first_pt = self.current_assignment.resource.
        _wp_sequence._sequence[0]
    own_posit = self._controller.get_own_state()
        .state.pose.pose.position.lat, \
            self._controller.get_own_state()
                .state.pose.pose.position.lon
    distance = gps.gps_distance(own_posit[0],

```

```

        own_posit[1], first_pt.lat, first_pt.lon)
    if distance <= 110:
        self.inside_search_box = True
        self.search_time_start = time.time()

    if self._wp_driver.is_complete():
        self.inside_search_box = False
        self.task_done()

    self.__ctr+= 1

    return self._wp_driver.get_waypoint()

elif self.__task_state == AuctionCue.IDLE:
    if self.__ctr == 0:
        self.aircraft_idle()
        prior_idling = str(round(self.
            time_idle_prior,2))

        if self.current_assignment == None:
            task_id = 'IDLE_BOT. AUCTIONS_COMPLETE'

        elif self.current_assignment != None:
            task_id = str(self.current_assignment.id
                )

        else:
            task_id = 'TASK_ID_UNKNOWN'

    self.data_compile.append("Null"+" "+str(
        task_id)+" "+str(self.final_auc_time)+" ")

```

```

+"N/A"+", "+ "N/A"+", "+ "N/A"+", "+str(
prior_idling))

for row in self.data_compile:
    self._controller.log_info(row)

if self.write_to_file == True:
    # Open File
    now = datetime.datetime.now()
    today = str(now.month) + '_' + str(now.
        day)
    name_of_file = 'ThesisOutput_Bot' + str(
        self._controller.own_id)+'_' + today

    completeName = os.path.join(self.
        save_path, name_of_file+".csv")

    resultFile = open(completeName, 'w')

    # Write data to file
    for row in self.data_compile:
        resultFile.write(row + "\n")

    resultFile.close()

    self._controller.log_info("OUTPUT_FILE_
        CREATED")

    output_list = []
    for f in os.listdir(self.save_path):
        if re.match('ThesisOutput', f):
            output_list.append(f)

```

```

if len(output_list) == len(self.
    _participants):

    now = datetime.datetime.now()
    today = str(now.month) + '_' + str(
        now.day)
    name_of_collated = '
        ThesisFinalResults_Bot' + str(
            self._controller.own_id)+'_' +
            today

    finalName = os.path.join(self.
        save_path , name_of_collated+".csv
        ")

    overallResult = open(finalName , 'w')

    self._controller.log_info("
        CONSOLIDATED_OUTPUT_COLLATING")

    for f in output_list:
        completeName = os.path.join(self
            .save_path , f)
        reading = open(completeName , "r")
        for line in reading:
            overallResult.write(line)
        reading.close()
    overallResult.close()

```

```

self.__ctr+=1
if self.__ctr in [1,2,3,4,5]:
    self._controller.log_info("\n*\n**\nBot is
        _IDLE\n**\n*")

return self._driver.get_waypoint()

elif self.__task_state == AuctionCue.INVESTIGATE:

    if self.__ctr == 15:
        self.notify_task_status(auctions.
            ResourceRecord.ACTIVE)
        self._auction.set_resource_status(self.
            current_assignment.id, auctions.
            ResourceRecord.ACTIVE)

    if self.current_assignment.resource.
        investigate_in_progress:
        if self.__ctr % 10 == 0:
            self._controller.log_info("time_
                remaining_on_station_is_" + str(int(
                    self.current_assignment.resource.
                        getTimeRemaining()))))

        if self.current_assignment.resource.is_done
            ():

                self.task_done()

        else:
            self.__ctr+=1
            return self._investigate_driver.

```

```

        get_waypoint()
    else:
        if self.__ctr % 50 == 0:

            own_posit = self._controller.
                get_own_state().state.pose.pose.
                position.lat, \
                    self._controller.
                        get_own_state().state
                        .pose.pose.position.
                        lon

            inv_pt = self.current_assignment.
                resource._investigate_pt[0],\
                    self.current_assignment.
                        resource.
                        _investigate_pt[1]

            dist = gps.gps_distance(own_posit[0],
                own_posit[1], inv_pt[0], inv_pt[1])

            self._controller.log_info("BOT_NOT_ON_
                STATION_YET_Remaining:" + str(int(
                dist)))

    if self.current_assignment.resource.
        on_station():
            self._controller.log_info("BOT_IS_ON_
                STATION_INVESTIGATION_IN_PROGRESS")

```

```

        self.__ctr+=1

        return self._investigate_driver.get_waypoint()

elif self.__task_state == AuctionCue.SEARCH_WAIT:

        self.time_idle_prior = time.time()-self.
            idle_start

        return self._wp_driver.get_waypoint()

def _start_next_task(self):
    self.current_assignment = self.next_assignment
    self.next_assignment = None
    self.detection_occurred = False
    self.wpt_ctr = 0
    self.__ctr = 0

    if self.current_assignment.is_null:
        self._controller.log_info(" Starting ▯ Null ▯ Task ▯ "
            + str(self.current_assignment.id))
        self.__task_state = AuctionCue.IDLE
        self.aircraft_idle()

    elif self.current_assignment.resource._task_id ==
        search.SEARCH_CELL:
        self._controller.log_info(" Starting ▯ Search ▯ Task ▯ "
            + str(self.current_assignment.id))

        self.__task_state = AuctionCue.SEARCH

        self.current_assignment.resource.set_task_wps((

```

```

        self._controller.get_own_state().state.pose.
        pose.position.lat, self._controller.
        get_own_state().state.pose.pose.position.lon)
    )

    self._wp_driver.reset(self.current_assignment.
        resource._wp_sequence)
    own_posit = self._controller.get_own_state().
        state.pose.pose.position.lat, \
        self._controller.
        get_own_state().state
        .pose.pose.position.
        lon

    return self._wp_driver.get_waypoint()

elif self.current_assignment.resource._task_id ==
    search.INVESTIGATE_PT:
        self._controller.log_info("Starting Investigate_
            Task_ " + str(self.current_assignment.id))

        self.__task_state = AuctionCue.INVESTIGATE

        self.current_assignment.resource.set_task_wps((
            self._controller.get_own_state().state.pose.
            pose.position.lat, self._controller.
            get_own_state().state.pose.pose.position.lon)
        )

        own_posit = self._controller.get_own_state().
            state.pose.pose.position.lat, \
            self._controller.
            get_own_state().state

```

```

        .pose.pose.position.
        lon

wpt = self.current_assignment.resource.
    _waypoints[0]

self._investigate_driver = transits.DirectDriver
    (self._controller, \
        wpt[0], \
        wpt[1], \
        self._controller.
            ap_wpt.z, \
        self._controller.
            ac_limits.
            nom_fwd_speed)

return self._investigate_driver.get_waypoint()

def process_behavior_data(self, data_msg):
    ''' Processes vehicle-to-vehicle SwarmBehaviorData
        messages
    @param data_msg: ap_msgs.SwarmBehaviorData object
        with message data
    '''
    if self.is_active():
        if self._auction:
            self._auction.process_message(data_msg)

    if data_msg.id == bytes.INDEX_USHORT:
        self._controller.log_info("RESOURCE_STATUS_
            UPDATE")
        parser = bytes.IndexUShortParser()
        parser.unpack(data_msg.params)

```

```

        self._auction.set_resource_status(parser.
            index, parser.value)

    elif data_msg.id == bytes.LAT_LON:
        self._controller.log_info("INVESTIGATION_
            TASK_ALERT!!!")
        parser = bytes.LatitudeLongitudeParser()
        parser.unpack(data_msg.params)
        position = (parser.latitude, parser.longitude
            )
        investigation_task = search.Investigate(self
            .investigation_id, self.investigation_time
            , position, self._controller)\

        self._resource_tracker.append("i")
        new_id = len(self._resource_tracker)-1
        self._controller.log_info("New_task_RECEIVED
            . There_are_now" + str(new_id) + " tasks
            ")
        self._auction._resources.add_resource(new_id
            , investigation_task)
        self.investigation_id += 1
        self.__auction_state = AuctionCue.RESTART

    elif data_msg.id == bytes.AUCTION_NEW:
        self._controller.log_info("NEW_AUCTION_
            MESSAGE_RECEIVED_AUCTION_RESTART")
        self.__auction_state = AuctionCue.RESTART

def notify_task_status(self, new_status):

    self._auction.set_resource_status(self.
        current_assignment.id, new_status)

```

```

        parser = bytes.IndexUShortParser()
        parser.index = self.current_assignment.id
        parser.value = new_status
        self._controller.directed_behavior_data_msg.params.
            id = bytes.INDEX_USHORT
        self._controller.directed_behavior_data_msg.params.
            params = parser.pack()

    for uav_id in self._participants:
        self._controller.directed_behavior_data_msg.
            destination = uav_id
        self._controller.behavior_data_to_publisher.
            publish(self._controller.
                directed_behavior_data_msg)

def new_auction(self):
    self._controller.directed_behavior_data_msg.params.
        id = bytes.AUCTION_NEW
    self._controller.directed_behavior_data_msg.params.
        params = b''
    for uav_id in self._participants:
        self._controller.directed_behavior_data_msg.
            destination = uav_id
        self._controller.behavior_data_to_publisher.
            publish(self._controller.
                directed_behavior_data_msg)
    self.__auction_state = AuctionCue.RESTART

def investigate_alert(self, uav_position):
    parser = bytes.LatitudeLongitudeParser()
    parser.latitude = uav_position[0]

```

```

        parser.longitude = uav_position[1]
        self._controller.directed_behavior_data_msg.params.
            id = bytes.LAT_LON
        self._controller.directed_behavior_data_msg.params.
            params = parser.pack()

    for uav_id in self._participants:

        self._controller.directed_behavior_data_msg.
            destination = uav_id
        self._controller.behavior_data_to_publisher.
            publish(self._controller.
                directed_behavior_data_msg)

    investigation_task = search.Investigate(self.
        investigation_id, self.investigation_time,
        uav_position, self._controller)
    self._resource_tracker.append("i")

    new_id = len(self._resource_tracker)-1
    self._controller.log_info("New task DETECTED. There
        are now" + str(new_id) + " tasks")
    self._auction._resources.add_resource(new_id,
        investigation_task)
    self.investigation_id += 1

    self.__auction_state = AuctionCue.RESTART

def drop_task(self):
    ''' when investigation alerts are received, if it is
        an AC_COPTER,
        they should drop all their tasks in order to bid on
        the investigation task

```

*unless they are 80% complete with their current task
or they are already
assigned to an investigate task. '''*

```

if self.aircraft_type != enums.AC_COPTER or self.
    __task_state == AuctionCue.INVESTIGATE:
    return

if self.current_assignment.resource._task_id ==
    search.SEARCH_CELL:
    if self.next_assignment.resource._task_id ==
        search.INVESTIGATE_PT:
        if self._wp_driver._sequence.
            percent_complete() < 85.0:
            self.notify_task_status(auctions.
                ResourceRecord.AVAILABLE)
            self.dropped_tasks += 1.0
            self._controller.log_info("Task_dropped_
                in_favor_of_investigation_task")
            self._start_next_task()

return

```

```

def task_done(self):
    self.notify_task_status(auctions.ResourceRecord.
        COMPLETE)
    prior_idling = str(round(self.time_idle_prior,2))
    auc_time = str(round(self.current_assignment.
        resource.time_to_win,2))
    transit_time = str( round(self.current_assignment.
        resource._task_transit_time,2) )

if self.current_assignment.resource._task_id ==

```

```

search.SEARCH_CELL:
    self.search_tasks_completed.append(self.
        current_assignment.resource._task_id)
    completed_id = str(self.current_assignment.id)
    search_time = str(round((time.time() - self.
        search_time_start),2) )
    self.data_compile.append('SEARCH'+','+
        completed_id+','+auc_time + ', ' +
        transit_time + ', ' + str(0.0) + ', ' +
        search_time + ', ' + prior_idling)

elif self.current_assignment.resource._task_id ==
search.INVESTIGATE_PT:
    self.investigate_tasks_completed.append(self.
        current_assignment.resource._task_id)
    completed_id = str(self.current_assignment.id)
    inv_time = str(float(self.current_assignment.
        resource.orbit_time))
    self.data_compile.append('INVESTIGATE'+','+
        completed_id+','+auc_time+','+transit_time+',
        '+inv_time+', '+str(0.0)+'+', '+prior_idling)

#when tasks finish their assignment the following
    attributes must be reset
self.current_assignment = None
self.detection_occurred = False
self.time_idle_prior = 0.0
self.inside_search_box = False

#if you have won a past auction and you have a next
    assignment, start it.
if self.next_assignment != None:

```

```

        self._controller.log_info("Next assignment is
            available in queue. Auction not required.")
        self._start_next_task()

#otherwise, start a new auction if you are not
    already in one
    else:
        self.__task_state = AuctionCue.SEARCH_WAIT
        self.idle_start = time.time()
        #if the auctions are done, be idle
        if self.__auction_state == AuctionCue.COMPLETE:
            self._controller.log_info("Auction not
                started because auctions complete.")
            #filename = "output" + str(self._controller.
                own_id) + ".csv"
            #data_collection = open(filename, 'w')
            #data_collection.write(Search Tasks
            self.__task_state = AuctionCue.IDLE
            self.aircraft_idle()

        else: # self.__auction_state != AuctionCue.
            IN_PROGRESS and self.__auction_state !=
            AuctionCue.RESTART:
            self.new_auction()

```

A.2 auctions Source Code

```

#
# Dr. Duane Davis, 2018
#

# Standard Python imports
import threading

```

```

import random

# ROS imports

# ARSENL imports
import arsenl_control.swarm_control as ctrl
import arsenl_library.math_utils as math_utils
import arsenl_behavior_tools.data_exchange as data_exchange

class ResourceRecord(object):
    ''' Class used to consolodate information about an
        auction resource
        (i.e., an item that can be bid for). Data within the
        object is not
        protected and can be modified as required by other
        objects and functions.

    Member variables:
        id: Integer ID of the resource
        agent_id: Integer ID of the current highest-bidding
            agent
        resource: object representation of the resource
        status: status of the object (e.g., available or
            complete)
        current_bid: most recent bid for the resource (
            successful or otherwise)
        utility: "value" of the resource to the bidding agent
        is_null: set to True for "placeholder" resources

    Member functions:
        copy: creates a (pseudo) deep copy of the object
        reset: resets variable values of the object

```

```

'''

# Resource status enumerations
AVAILABLE = 0 # Resource available for bid
ASSOCIATED = 1 # Resource associated with an agent but
    still up for bid
ASSIGNED = 2 # Resource has been assigned to an agent
    but still up for bid
ACTIVE = 3 # Resource has been assigned to an agent
    but not up for bid
COMPLETE = 4 # Resource has been completed or
    exhausted and not up for bid
ALL_STATUS = { AVAILABLE, ASSOCIATED, ASSIGNED, ACTIVE,
    COMPLETE }
AVAIL_STATUS = { AVAILABLE, ASSOCIATED, ASSIGNED }

def __init__(self, resource_id, resource_object):
    ''' Initializer for the class sets up class
        variables
    @param resource_id: unique identifier for the
        resource
    @param resource_object: object representation of the
        resource
    '''
    self.id = resource_id
    self.resource = resource_object
    self.agent_id = 0
    self.status = ResourceRecord.AVAILABLE
    self.current_bid = 0.0
    self.utility = 0.0
    self.is_null = False
    self.search_task = True

```

```

def __repr__(self):
    ''' Generates a string representation of the
        ResouceRecord
    '''

    return "Resource_ID:_%d, Utility:_%f, Status:_%d,
        Agent:_%d, Bid:_%f" \
            %(self.id, self.utility, self.status, self.
              agent_id, self.current_bid)

def copy(self):
    ''' Instantiates a pseudo-deep copy of the resource
        (the resource
        object is copied by reference, but other members are
        copied by value
        @return a duplicate object of the ResrouceRecord
    '''

    new_record = ResourceRecord(self.id, self.resource)
    new_record.agent_id = self.agent_id
    new_record.status = self.status
    new_record.current_bid = self.current_bid
    new_record.utility = self.utility
    new_record.is_null = self.is_null
    new_record.search_task = self.search_task
    return new_record

def set_investigate(self):
    self.search_task = False

def reset(self):

```

```
''' Resets variable values to initial settings  
, , ,
```

```
self.agent_id = 0  
self.status = ResourceRecord.AVAILABLE  
self.current_bid = 0.0  
self.utility = 0.0
```

```
class ResourceSet(object):
```

```
''' Class used to encapsulate a set of resources that  
will be  
used in a single-item auction (can be extended to  
account for  
multi-item auctions if required).
```

Member variables:

```
--resources: dictionary of ResourceRecord objects  
--lock: re-entrant lock object to prevent concurrent  
access
```

Member functions:

```
copy: returns a duplicate of the object  
keys: returns a list of resource keys (IDs)  
resources: returns a list of resources (ResourceRecord  
objects)  
num_available_keys: returns the number of available-  
for-bid resources  
available_keys: returns a list of resource keys that  
are available for bid  
available_resources: returns a list of resources that  
are available for bid  
add_resource: used to add a resource to the set of  
auction resources
```

```

    clear: used to erase everything and start over
    reset: used to reset all resources to their original
        values
    get: used to return a reference to a resource
    remove: used to remove a resource
    , , ,

```

```

def __init__(self):
    ''' Initializer for the class sets up class
        variables
    , , ,

    self.__resources = dict()
    self.__lock = threading.RLock()

```

```

def __repr__(self):
    ''' Prints a string representation of the
        ResourceSet
    , , ,

    result = ""
    count = 0
    with self.__lock:
        for key in self.__resources.keys():
            count += 1
            result += str(self.__resources[key])
            if count < len(self.__resources):
                result += ", "
    return result

```

```

def copy(self):
    ''' Creates a deep copy of the object
        @return a duplicate ResourceSet

```

```

    , , ,

new_set = ResourceSet()
with self.__lock:
    for resource_id , resource in self.__resources.
        items():
            new_set.add_resource(resource_id , resource.
                copy())
    return new_set

def clear(self):
    ''' Deletes everything associated with the resource
        set
    , , ,

    with self.__lock:
        self.__resources.clear()

def reset(self):
    ''' Resets the bid and local utility values for all
        resources
    , , ,

    with self.__lock:
        for record in self.__resources.values():
            record.reset()

def keys(self):
    ''' Returns a list of resource keys (IDs)
    , , ,

    with self.__lock:
        return self.__resources.keys()

```

```

def resources(self):
    ''' Returns a list of resources (ResourceRecord
        objects)
    '''
    with self.__lock:
        return self.__resources.values()

def num_available_keys(self):
    return len(self.available_keys())

def available_keys(self):
    ''' Returns a list of resource keys that are
        available for bid
        (i.e., neither in progress nor complete)
        @return list of resource_ids that are available for
            bid
    '''
    result = []
    with self.__lock:
        for key in self.__resources.keys():
            if self.__resources[key].status in
                ResourceRecord.AVAIL_STATUS:
                result.append(key)
    return result

def available_resources(self):
    ''' Returns a dictionary of resources that are
        available for bid
        (i.e., neither in progress nor complete)

```

```

        @return dictionary of available-for-bid resources
        '''
        result = dict()
        with self.__lock:
            for resource_id in self.__resources.keys():
                resource = self.__resources[resource_id]
                if resource.status in ResourceRecord.
                    AVAIL_STATUS:
                        result[resource_id] = resource
        return result

def add_resource(self, resource_id, resource_object):
    ''' Adds an initialized resource to the set
    @param resource_id: unique identifier for the
        resource
    @param resource_object: object representation of the
        resource
    '''

    # If already a ResourceRecord add it as is; if not
    # instantiate one
    with self.__lock:
        if isinstance(resource_object, ResourceRecord):
            self.__resources[resource_id] =
                resource_object
        else:
            resource = ResourceRecord(resource_id,
                resource_object)
            self.__resources[resource_id] = resource

def get(self, resource_id):

```

```

''' Used to retrieve a reference to a resource
    record
@param resource_id: id of the resource being
    retrieved
@return the retrieved resource record
'''
with self.__lock:
    return self.__resources[resource_id]

def remove(self, resource_id):
    ''' Used to remove a resource record from the set
@param resource_id: id of the resource to be removed
@return a reference to the removed resource record
'''
with self.__lock:
    return self.__resources.pop(resource_id)

def set_status(self, resource_id, status):
    ''' Set the status of a resource to the specified
        value
@param resource_id: identification of the resource
        to be updated
@param status: integer enumeration for the new
        status
'''
if status not in ResourceRecord.ALL_STATUS:
    raise Exception("Attempt to set resource to
        invalid status: %d"%status)
with self.__lock:
    try:
        self.__resources[resource_id].status =

```

```

        status
    except KeyError:
        pass # probably a since-removed null task

def add_null_resources(self, num_participants):
    ''' Adds null resources so that the number of
        available-for-bid
        resources is greater than or equal to the number of
        auction
        participants (reduces bidding wars for few remaining
        resources)
    @param num_participants: number of participating
        agents
    '''
    with self.__lock:
        self.__remove_null_resources() # start with a
            clean slate
        num_resources = self.num_available_keys()
        num_nulls = num_participants - num_resources
        if num_nulls > 0 and num_resources > 0:
            max_id = max(self.__resources.keys())
            for null_id in range(max_id + 1, max_id + 1
                + num_nulls):
                null_resource = ResourceRecord(null_id,
                    None)
                null_resource.is_null = True
                self.__resources[null_id] =
                    null_resource

def __remove_null_resources(self):
    ''' Removes null resources from the set

```

```

    , , ,

    for key in self.__resources.keys():
        if self.__resources[key].is_null:
            self.__resources.pop(key)

```

class SingleItemAuction(**object**):

''' Class for implementing a single-item auction for task or resource allocation among swarm vehicles. The generic class can be used as-is with an behavior providing and manipulating the application-specific information, or it can be extended to fully implement specific auctions.

Member variables:

- _own_id: identification of this agent in the auction*
- _controller: reference to the behavior control object (used for logging)*
- _epsilon: minimum bid increment*
- _resources: resource set object containing auction resource records*
- _working_resource_set: working set of resources to maintain consistency*
- _participants: set of auction participant IDs*
- __utility_function: computes a utility value given a resource record*
- __round: current bidding round of the auction*
- __complete: set to True once the current auction has completed*
- __in_progress: set to True once the auction has commenced*

_assignments: dictionary of agent_id-to-resource mappings
_own_bids: ordered list of per-round bids by the agent
_this_rnd_bid_collector: consensus object that collects current-round bids
_last_rnd_bid_collector: consensus object that collects previous-round bids

Member functions:

initialize_auction: initializes the auction to run for the first time
continue_auction_from_current: starts a new auction from the current state
auction_step: runs one iteration of the auction process
set_local_utility_function: sets the function for evaluating local utilities
set_participants: resets the set of participants
set_resources: initializes the resources for the auction
add_resources: adds resources to the existing set
set_resource_status: used to update the status of a specific resource
round: "get" method for the current auction round
complete: "get" method for the current auction completion status
assignments: "get" method for the completed auction's assignments
make_next_bid: safely increments to the next round and places the bid
send_bid_info: sends bid information to other participants as required
process_message: processes SwarmBehaviorData messages

```

    __update_local_utilities: calculates local resource
        utility values
    __synchronize_and_update: determines when it is time
        to increment rounds
    __generate_bid: determines the bid for the next round
    __auction_complete: determines when the auction is
        complete
'''

def __init__(self, own_id, controller=None):
    ''' Initializer for the class declares and sets
        member variables
    @param own_id: iterable object of auction
        participant IDs
    @param controller: reference to the owning
        SwarmControl object
    '''
    self._own_id = own_id
    self._controller = controller
    self._epsilon = 0
    self._resources = ResourceSet()
    self._working_resource_set = None
    self._participants = { self._own_id }
    self.__utility_function = None
    self.__round = 0
    self.__complete = False
    self.__in_progress = False
    self._assignments = None
    self._own_bids = dict()
    self._this_rnd_bid_collector = None
    self._last_rnd_bid_collector = None

```

```

def set_controller(self , controller):
    ''' Sets the reference to the behavior-owning
        SwarmControl object
        (primarily included to facilitate logging)
        @param controller: reference to the owning
        SwarmControl object
    '''
    self._controller = controller


def set_local_utility_function(self , func):
    ''' Sets the function for calculating resource local
        utility values.
        The function should take a single ResourceRecord
        parameter and
        calculate an appropriate local utility value for the
        owning agent.
        @param func: auction-specific local utility
        calculation function
    '''

    self.__utility_function = func


def set_participants(self , participants):
    ''' Resets the set of participants. Deletes old
        participant
        records that are not in the new participant set and
        adds
        new participant records for those that were not in
        the old set
        @param participants: list of participant IDs
    '''

```

```

if self._own_id not in participants:
    raise KeyError("Own_ID_must_be_included_in_
        auction_participants")

self._participants = set(participants)


def set_resources(self, resources):
    ''' Sets the set of resources to the parameter
        values
    @param resources: list of (id, resource) tuples to
        be added
    '''
    self._resources.clear()
    for resource in resources:
        self._resources.add_resource(resource[0],
            resource[1])


def add_resources(self, resources):
    ''' Adds resource records to the existing set of
        resources
    @param resources: list of (id, resource) tuples to
        be added
    '''
    for resource in resources:
        self._resources.add_resource(resource[0],
            resource[1])


def set_resource_status(self, resource_id, status):
    ''' Used to update the status of a resource (e.g.,

```

```

        from
        ASSIGNED to IN_PROGRESS or IN_PROGRESS to COMPLETE)
        @param resource_id: ID of the resource to be updated
        @param status: enumeration (int) for the updated
            states
        '''
        self._resources.set_status(resource_id, status)

def complete(self):
    ''' @return True if the current auction is complete
    '''
    return self.__complete

def assignments(self):
    ''' @return the auction-derived assignments (or None
        if not complete)
    '''
    if self.__complete:
        return self._assignments
    return None

def resource_status(self, resource_id):
    ''' Returns the current status of a selected
        resource
        Raises KeyError if the requested resource_id is not
        in the set
        @param resource_id: ID of the resource for which
            status is requested
        @return status (enumeration) of the resource
    '''

```

```

    return self._resources.get(resource_id).status

def round(self):
    ''' @return the current auction round
    '''
    return self.__round

def make_next_bid(self, resource_id, bid):
    ''' Increments to the next round and sets up the bid
        collector
    @param resource_id: ID of the resource that this
        agent will bid on
    @param bid: bid (value) for the desired resource
    '''
    self.__round += 1
    (self._last_rnd_bid_collector, self.
     _this_rnd_bid_collector) = \
        (self._this_rnd_bid_collector, self.
         _last_rnd_bid_collector)
    self._own_bids[self.__round] = (resource_id, bid)
    self._this_rnd_bid_collector.reset((self.__round,
        resource_id, bid))

def initialize_auction(self, epsilon, resources,
    participants):
    ''' Initializes an auction to run (initializes the
        resource set and
        other internal variables to ensure correctness at
        the start)
    @param epsilon: minimum bid increase value per round

```

```

    @param resources: list of (id, resource) tuples to
        be auctioned
    @param participants: list of auction participant IDs
    @return True if ready to commence (False mean no
        available resources)
    ...

    self.__round = 0
    self.__complete = False
    self.__in_progress = False
    self._epsilon = abs(epsilon)
    self._assignments = None
    self._own_bids.clear()
    self.set_resources(resources)
    self.set_participants(participants)
    self._resources.add_null_resources(len(participants)
        )
    self._this_rnd_bid_collector = \
        data_exchange.LossyAuctionBidExchange(
            participants, \

            self.
                _controller
            .
                crashed_keys
            , \
        self.
            _controller
            .locks[
                ctrl.
                    SwarmController
                    .SWARM
                ], \
        self.
            _own_id

```

```

)
self._this_rnd_bid_collector.set_msg_publisher(self.
    _controller.behavior_data_publisher)
self._last_rnd_bid_collector = \
    data_exchange.LossyAuctionBidExchange(
        participants , \
            self.
                _controller
                .
                crashed_keys
                , \
            self.
                _controller
                .locks[
                ctrl.
                SwarmController
                .SWARM
                ], \
            self.
                _own_id
            )
self._last_rnd_bid_collector.set_msg_publisher(self.
    _controller.behavior_data_publisher)
self.__update_local_utilities()
self._working_resource_set = self._resources.copy()
return (self._resources.num_available_keys() > 0)

```

```

def continue_auction_from_current(self):
    ''' Resets the auction to start from the current
        state so that
        the next set of resources can be allocated
        @return True if ready to commence (False mean no

```

```

        available resources)
    , , ,

    self.__round = 0
    self.__complete = False
    self.__in_progress = False
    self._assignments = None
    self._own_bids.clear()
    self._resources.add_null_resources(len(self.
        _participants))
    self.__update_local_utilities()
    self._working_resource_set = self._resources.copy()
    return (self._resources.num_available_keys() > 0)

def auction_step(self):
    ''' Runs one iteration of the auction algorithm.
        Ensures
        synchronization, places the bid, updates resource
        statuses based
        on the current bids, determines assignments based on
        the bids.
        @return True when the auction is complete (None
        otherwise)
    , , ,

    #if it isn't synced up, dont do anything
    if not self.__synchronize_and_update():
        return None

    # either return the task that the bot was assigned
    to
    if self.__auction_complete():
        return self._assignments[self._own_id]

```

```

        # Or generate a new bid
        bid = self.__generate_bid()
        self.make_next_bid(bid[0], bid[1])

    return None

def send_bid_info(self):
    ''' Sends bid info to other vehicles as required
    '''

    if self.__round > 1:
        self._last_rnd_bid_collector.send_requested()
        self._this_rnd_bid_collector.send_requested()

def process_message(self, msg):
    ''' Process a received SwarmBehaviorData network
        message
        Should be called from owning behavior's
        process_behavior_data method
        @param msg: received swarm behavior data message
    '''

    if self._last_rnd_bid_collector:
        self._last_rnd_bid_collector.process_message(msg
        )
    if self._this_rnd_bid_collector:
        self._this_rnd_bid_collector.process_message(msg
        )

def __update_local_utilities(self):
    ''' Updates resource local utility values
    '''

```

```

for resource_id, resource in self._resources.
    available_resources().items():
        self.set_resource_status(resource_id,
            ResourceRecord.AVAILABLE)
        if not resource.is_null:
            resource.utility = self.__utility_function(
                resource)

def __synchronize_and_update(self):
    ''' Uses current round bid data to associate
        resources with the
        highest-bidding agents.
    @return True if it's time to proceed to the next
        round
    '''
    # Just start the auction if it hasn't started yet
    if not self.__in_progress:
        self.__in_progress = True
        return True

    self.send_bid_info()

    bids = self._this_rnd_bid_collector.get_results()
    if not bids:      # Don't have all bids for this round
        return None

    # Bids are a tuple of (agent (round, resource, bid))
    tuples
    for bid_record in bids:
        agent_id = bid_record[0]
        resource_id = bid_record[1][1]

```

```

        bid = bid_record[1][2]

        # Associate the resource with the bidder if bid
        is now highest
        resource = self._working_resource_set.get(
            resource_id)
        if (resource.status in ResourceRecord.
            AVAIL_STATUS):
            if bid > resource.current_bid:
                resource.current_bid = bid
                resource.agent_id = agent_id
                resource.status = ResourceRecord.
                    ASSOCIATED
            elif (bid == resource.current_bid) and \
                (agent_id < resource.agent_id):
                resource.current_bid = bid
                resource.agent_id = agent_id
                resource.status = ResourceRecord.
                    ASSOCIATED

        # TODO: Add in crashed agent functionality for in-
        progress cells
        return True

def __generate_bid(self):
    ''' Determines the desired resource and bid price to
        be posted for
        the current round
        @return (round, resource_id, bid_price) tuple
    '''
    # If last bid won (for now), resubmit it
    if self.__round > 0:

```

```

last_bid = self._own_bids[self.__round]
resource = self._working_resource_set.get(
    last_bid[0])
if resource.agent_id == self._own_id:

    bid = (resource.id, resource.current_bid)
    self._controller.log_info("Previous bid
        still valid for round %d, resubmitting %s"
        "\
                                %(self.__round+1,
                                    str(bid)))

    return bid

# New bid is difference between best & 2nd-best
    utility + epsilon
resources = sorted(self._working_resource_set.
    available_resources().values(), \
                    key = lambda x: x.utility - x.
                        current_bid, \
                    reverse = True)

bid_increase = resources[0].utility - \
    resources[0].current_bid + self.
        _epsilon
if len(resources) > 1:
    bid_increase -= (resources[1].utility - \
                    resources[1].current_bid)
bid_value = resources[0].current_bid + \
    max(bid_increase, self._epsilon)
bid = (resources[0].id, bid_value)

```

```

self._controller.log_debug("Calculated the following
    bid for round %d: %s" \
                            %(self.__round+1, str(bid)
                                ))
return bid

def __auction_complete(self):
    ''' Determines whether or not the auction has
        completed (i.e.,
        all agents have bid for a different resource).
        Updates the
        assignments and resource statuses if and only if the
        auction
        has been determined to be completed.
        @return True if the auction is complete
        , , ,

    if self.__complete:
        return True
    associated = dict()
    participating = self._participants.difference(self.
        _controller.crashed_keys)

    # Identify all of the resources that are "associated
    #   " with an agent
    for resource_id in self._working_resource_set.keys():
        resource = self._working_resource_set.get(
            resource_id)
        if resource.status == ResourceRecord.ASSOCIATED:
            associated[resource.agent_id] = self.
                _resources.get(resource_id)

```

```

        # Done if the # of "associated" resources equals the
        # of agents
        if len(associated.keys()) == len(participating):
            self._assignments = associated
            self._controller.log_debug("Auction complete in %
                d rounds with the following assignments: %s" \
                                     %(self.__round, self.
                                         _assignments))

            self.__complete = True
            return True
        return False

```

A.3 searcher Source Code

```

#
# Major Matthew Hopchak 2018
# LT Britt Campbell 2019
#

# Standard python imports
import math
import time

# ROS imports
import rospy

# ARSENL imports
import arsenl_control.swarm_control as ctrl
import arsenl_control.plugin_behavior as plugin
import arsenl_behavior_tools.auctions as auctions
import arsenl_algorithms.independent_transits as transits
import arsenl_behavior_tools.waypoint_paths as paths

```

```

# Utility imports
import arsenal_library.math_utils as ro_math
import arsenal_library.gps_utils as gps
import arsenal_library.bitmapmapped_bytes as bytes

# Task enumerations
SEARCH_CELL = 1
INVESTIGATE_PT = 2

class SearchAreaPartition(plugin.PluginBehavior):

    # Class-specific enumerations and constants
    # Basic rectangular search area enumerations (no
        obstacles)
    BASIC_LIVE_FLY = 0
    BASIC_LARGER    = 1
    # Complex polygonal search area enumeration (with
        obstacles)
    COMPLEX = 2

    # Search area southwest location
    AREA_SW_LAT = 35.721147    # these values will be
        modified when generate search area is called
    AREA_SW_LON = -120.773008

    def __init__(self, choose_search_area, controller):
        self._area_choice = choose_search_area
        self._controller = controller
        #cells remaining
        self.num_cells = 0
        self._cells_left    = set()
        #dictionary holding all the cells

```

```

self._cells          = { }
#boundaries of the search area
self._west_wall      = [ ]
self._north_wall     = [ ]
self._east_wall      = [ ]
self._south_wall     = [ ]
#the southwest corner needs to be globally
accessible
self.area_sw_lat = 35.721147 # these values will
be modified when generate search area is called
self.area_sw_lon = -120.773008
self.area_orient = 0
self.area_nw_lat = 0
self.area_nw_lon = 0
self.area_se_lat = 0
self.area_se_lon = 0

self._sensor_sweep = [75, 75]

```

```

def generateBasicCells(self, area_length, area_width,
c_length, c_width):
    ''' creates cell objects of rectangular shape of
        specified height/width (m)
        Cells will be integer numbered starting at 0.
    '''

    # Define cell dimensions
    l_divisor = int(math.ceil(area_length / c_length))
    w_divisor = int(math.ceil(area_width / c_width))
    cell_length = area_length / l_divisor
    cell_width = area_width / w_divisor
    self._controller.log_info("l_divisor:␣" + str(
        l_divisor))

```

```

self._controller.log_info( "area_length:_" + str(
    area_length) + "\ncell_length:_" + str(
    cell_length) + "\narea_width:_" + str(area_width)
    + "\ncell_width:_" + str(cell_width) + "\n_
    ceiling(arealength/c_length)_=" + str(l_divisor)
    )

```

```

## print out the lat lons of the cells for box
drawing purposes

```

```

top_point_lat = self.area_nw_lat
top_point_lon = self.area_nw_lon

```

```

bottom_point_lat = self.area_sw_lat
bottom_point_lon = self.area_sw_lon

```

```

left_point_lat = self.area_sw_lat
left_point_lon = self.area_sw_lon

```

```

right_point_lat = self.area_se_lat
right_point_lon = self.area_se_lon

```

```

for i in range(l_divisor-1):
    left_point = gps.gps_newpos(left_point_lat ,
        left_point_lon , self.area_orient , cell_length
    )
    right_point = gps.gps_newpos(right_point_lat ,
        right_point_lon , self.area_orient ,
        cell_length)

```

```

self._controller.log_info("left:_" + str(
    left_point) + "right:_" + str(right_point) )

```

```

left_point_lat = left_point[0]
left_point_lon = left_point[1]

right_point_lat = right_point[0]
right_point_lon = right_point[1]

for i in range(w_divisor-1):
    top_point = gps.gps_newpos(top_point_lat ,
                               top_point_lon , self.area_orient + math.pi
                               /2.0, cell_width)
    bottom_point = gps.gps_newpos(bottom_point_lat ,
                                   bottom_point_lon , self.area_orient + math.pi
                                   /2.0, cell_width)

    top_point_lat = top_point[0]
    top_point_lon = top_point[1]

    bottom_point_lat = bottom_point[0]
    bottom_point_lon = bottom_point[1]

# Make these modified values publicly available
self.num_cells = l_divisor * w_divisor
self._controller.log_info("Number_of_cells_that_
    should_be_created:_" + str(self.num_cells))

# Define length_lines as guidelines for cell
    boundaries along the length of the area
num_length_lines = l_divisor - 1
length_lines = [ ]
chalk_line     = [ ]

```

```

length_lines.append(self._south_wall)
for i in range(0, num_length_lines):
    temp = length_lines[-1]
    chalk_line = [ (temp[0][0] + cell_length, temp
                    [0][1]), \
                    (temp[1][0] + cell_length, temp
                    [1][1]) ]
    length_lines.append(chalk_line)
length_lines.append(self._north_wall)
# Define width_lines as guidelines for cell
boundaries along the width of area
num_width_lines = w_divisor - 1
width_lines = [ ]
chalk_line = [ ]
width_lines.append(self._west_wall)
for i in range(0, num_width_lines):
    temp = width_lines[-1]
    chalk_line = [ (temp[0][0], temp[0][1] +
                    cell_width), \
                    (temp[1][0], temp[1][1] +
                    cell_width) ]
    width_lines.append(chalk_line)
width_lines.append(self._east_wall)

cell_id, j, k = 0, 0, 0
temp_cell = None

while cell_id < self.num_cells:
    for r in range(0, num_length_lines + 1):
        # cell south-west corner
        sw_width_norm = ro_math.
            normal_form_parameters(width_lines[k][0],
                                   width_lines[k][1])

```

```

sw_length_norm = ro_math.
    normal_form_parameters( length_lines[j]
        ][0], length_lines[j][1])
sw = ro_math.line_intersect( sw_width_norm
    [1], sw_width_norm[0], \
                                sw_length_norm
                                [1],
                                sw_length_norm
                                [0] )

# cell north-west corner
j += 1
nw_width_norm = ro_math.
    normal_form_parameters( width_lines[k][0],
        width_lines[k][1])
nw_length_norm = ro_math.
    normal_form_parameters( length_lines[j]
        ][0], length_lines[j][1])
nw = ro_math.line_intersect( nw_width_norm
    [1], nw_width_norm[0], \
                                nw_length_norm
                                [1],
                                nw_length_norm
                                [0] )

# cell north-east corner
k += 1
ne_width_norm = ro_math.
    normal_form_parameters( width_lines[k][0],
        width_lines[k][1])
ne_length_norm = ro_math.
    normal_form_parameters( length_lines[j]
        ][0], length_lines[j][1])
ne = ro_math.line_intersect( ne_width_norm
    [1], ne_width_norm[0], \

```

```

ne_length_norm
    [1],
    ne_length_norm
    [0] )

# cell south-east corner
j -= 1
se_width_norm = ro_math.
    normal_form_parameters( width_lines[k][0],
        width_lines[k][1])
se_length_norm = ro_math.
    normal_form_parameters( length_lines[j
    ][0], length_lines[j][1])
se = ro_math.line_intersect( se_width_norm
    [1], se_width_norm[0], \
        se_length_norm
        [1],
        se_length_norm
        [0] )

# Create Cell object
temp_cell = Cell(cell_id, [(sw, nw), (nw, ne
    ), (ne, se), (sw, se)], self._controller)
temp_cell.set_value(cell_length, cell_width)

self._cells[cell_id] = temp_cell
self._cells[cell_id].setWestBound((sw, nw))
self._cells[cell_id].setEastBound((ne, se))
self._cells_left.add(cell_id)
k -= 1
j += 1
self._controller.log_info("cell_" + str(
    cell_id) + "_created")
cell_id += 1

```

```

        self._controller.log_info("NEW_LINE")
        k += 1
        j = 0
    self._controller.log_info("\n*\n**\n_---" + str(
        cell_id) + "_Cells_successfully_partitioned.---_\n**\n*")

def generateBasicSearchArea(self):
    ''' fills boundary data structures given basic,
        large, or complex area
    '''

    if self._area_choice == SearchAreaPartition.
        BASIC_LIVE_FLY:
        cell_length = 200.0
        cell_width  = 200.0

        self.area_sw_lat = 35.721147
        self.area_sw_lon = -120.773008
        self.area_orient = 25.183537917993224
        area_length = 575.0
        area_width  = 750.0

    elif self._area_choice == SearchAreaPartition.
        BASIC_LARGER:

        cell_length = 250.0
        cell_width  = 250.0
        self.area_sw_lat = 35.721147
        self.area_sw_lon = -120.773008
        self.area_orient = 0
        area_length = 1200.0

```

```

        area_width = 1950.0

    elif self._area_choice == SearchAreaPartition.COMPLEX:
        self.area_sw_lat = 35.72102
        self.area_sw_lon = -120.79111
        self.area_orient = 0
        area_length = 2300.0
        area_width = 3000.0

    self._search_area = gps.GeoBox(self.area_sw_lat,
        self.area_sw_lon, area_length, area_width, self.
        area_orient)
    corners = self._search_area._cart_corners
    self._north_wall = (corners[1], corners[2])
    self._south_wall = (corners[0], corners[3])
    self._west_wall = (corners[0], corners[1])
    self._east_wall = (corners[2], corners[3])
    self._controller.log_info("\n*\n**\nOuter_search_
        area_boundary_generated.\n**\n*")

    nw_corner = gps.gps_newpos(self.area_sw_lat, self.
        area_sw_lon, self.area_orient, area_length)
    se_corner = gps.gps_newpos(self.area_sw_lat, self.
        area_sw_lon, self.area_orient + math.pi/2.0,
        area_width)
    ne_corner = gps.gps_newpos(se_corner[0], se_corner
        [1], self.area_orient, area_length)

    self._controller.log_info("\n*\n**\nNW_CORNER:_"
        + str(nw_corner) + "\n**\n*")
    self._controller.log_info("\n*\n**\nSE_CORNER:_"

```

```

        + str(se_corner) + "\n**\n*")
self._controller.log_info("\n*\n**\n\nNE_CORNER: "
        + str(ne_corner) + "\n**\n*")

self.area_nw_lat = nw_corner[0]
self.area_nw_lon = nw_corner[1]

self.area_se_lat = se_corner[0]
self.area_se_lon = se_corner[1]

if self._area_choice in [SearchAreaPartition.
    BASIC_LIVE_FLY, SearchAreaPartition.BASIC_LARGER
]:
    self.generateBasicCells(area_length, area_width,
        cell_length, cell_width)

class Investigate (object):
    """
    Class for maintaining attributes of each detection event
    found by a bot.
    Each point represents a biddable resource in Auctions.py
    that was inserted by a bot.
    """

    def __init__(self, investigation_id, time, position,
        controller):
        self._investigation_id = investigation_id
        self._task_value = 600.0
        self._controller = controller
        self._investigate_pt = position[0], position[1]
        self._west_bound = [ ]
        self._east_bound = [ ]
        self._waypoints = [ ]

```

```

        self._task_id = INVESTIGATE_PT
        self.lastWaypoint = self._investigate_pt
        self.orbit_time = time
        self.investigate_in_progress = False
        self._task_transit_time = 0.0
        self.time_to_win = 0.0

    def set_value(self, time):
        self._task_value = 600.0
        self.orbit_time = time

    def getTimeToAccomplishTask(self, spd):
        return self.orbit_time

    def getTimeRemaining(self):
        if self.investigate_in_progress:
            time_remaining = self.orbit_time - (time.time()
            - self.time_start)
            return time_remaining
        else:
            return self.orbit_time

    def getTaskTransitTime(self, last_wp_loc):
        entry_pt = self._investigate_pt
        spd = self._controller.ac_limits.nom_fwd_speed
        distance = gps.gps_distance(last_wp_loc[0],
            last_wp_loc[1], entry_pt[0], entry_pt[1])
        self._task_transit_time = distance/spd

        return self._task_transit_time

    def set_task_wps(self, uav_location):
        self._waypoints = [ ]

```

```

        self._waypoints.append(self.lastWaypoint)

    def is_done(self):
        if self.investigate_in_progress:
            time_on_station = time.time()-self.time_start
            if time_on_station < self.orbit_time:
                return False
            else:
                return True
        else:
            return False

    def on_station(self):
        uav_loc = self._controller.get_own_state().state.
            pose.pose.position.lat, \
                self._controller.get_own_state().state.
                    pose.pose.position.lon
        on_station_check = gps.gps_distance(uav_loc[0],
            uav_loc[1], self._investigate_pt[0], self.
                _investigate_pt[1])
        if on_station_check <= 75:
            self.investigate_in_progress = True
            self.time_start = time.time()
            return True
        else:
            return False

class Cell(object):
    """
    Class for maintaining attributes of each cell created by
    decomposition.
    Each Cell represents a biddable resource in Auctions.py.
    """

```

```

AREA_SW_LAT = 35.721147
AREA_SW_LON = -120.773008

```

```

def __init__(self, cell_id, boundary, controller):
    self._cell_id      = cell_id
    self._task_value   = 250.0
    self._boundary     = boundary
    self._controller   = controller
    self._west_bound   = [ ]
    self._east_bound   = [ ]
    self._waypoints    = [ ]
    self._task_id      = SEARCH_CELL
    self._trajectory    = paths.Trajectory()
    self._wp_sequence  = paths.WaypointSequencer()
    self.lastWaypoint  = None
    self._sensor_sweep = 50.0
    self._size          = 0.0
    self.time_to_win   = 0.0
    self._task_transit_time = 0.0

def set_value(self, length, width):
    self._task_value = length+width -130.0

def getWaypoints(self):
    return self._waypoints

def getCorners(self):
    '''returns the cartesian corners in the order sw, nw
    , ne, se'''

    corners = []
    SW_corner = self._west_bound[0]
    #corners.append(SW_corner)

```

```

NW_corner = self._west_bound[1]
#corners.append(NW_corner)
NE_corner = self._east_bound[0]
#corners.append(NE_corner)
SE_corner = self._east_bound[1]
#corners.append(SE_corner)
corners = [ ("SW",SW_corner) ,("NW",NW_corner) ,("NE",
            NE_corner) ,("SE",SE_corner)]

return corners


def getTimeToAccomplishTask(self , last_wp_loc):
    #returns the time to go between each of the
    waypoints
    spd = self._controller.ac_limits.nom_fwd_speed
    self.set_task_wps(last_wp_loc)
    total_cell_distance = self._wp_sequence.distance()
    task_accomplish_time = total_cell_distance/spd
    return task_accomplish_time


def getTimeRemaining(self):
    spd = self._controller.ac_limits.nom_fwd_speed
    remaining_cell_distance = self._wp_sequence.
        remaining_distance()
    remaining_time = remaining_cell_distance/spd
    return remaining_time


def getTaskTransitTime(self , last_wp_loc):

    spd = self._controller.ac_limits.nom_fwd_speed
    cell_corners = self.getCorners()

    SW_corner_cart = cell_corners[0][1]

```

```

NW_corner_cart = cell_corners[1][1]
NE_corner_cart = cell_corners[2][1]
SE_corner_cart = cell_corners[3][1]

SW_corner_gps = gps.gps_offset(Cell.AREA_SW_LAT,
                                Cell.AREA_SW_LON, \
                                SW_corner_cart
                                [1],
                                SW_corner_cart
                                [0])
NW_corner_gps = gps.gps_offset(Cell.AREA_SW_LAT,
                                Cell.AREA_SW_LON, \
                                NW_corner_cart
                                [1],
                                NW_corner_cart
                                [0])
NE_corner_gps = gps.gps_offset(Cell.AREA_SW_LAT,
                                Cell.AREA_SW_LON, \
                                NE_corner_cart
                                [1],
                                NE_corner_cart
                                [0])
SE_corner_gps = gps.gps_offset(Cell.AREA_SW_LAT,
                                Cell.AREA_SW_LON, \
                                SE_corner_cart
                                [1],
                                SE_corner_cart
                                [0])

bring_in_distance = 50.0
SW_bring_in_direction = gps.gps_bearing(
    SW_corner_gps[0], SW_corner_gps[1], NE_corner_gps
    [0], NE_corner_gps[1])

```

```

NE_bring_in_direction = gps.gps_bearing(
    NE_corner_gps[0], NE_corner_gps[1], SW_corner_gps
    [0], SW_corner_gps[1])
NW_bring_in_direction = gps.gps_bearing(
    NW_corner_gps[0], NW_corner_gps[1], SE_corner_gps
    [0], SE_corner_gps[1])
SE_bring_in_direction = gps.gps_bearing(
    SE_corner_gps[0], SE_corner_gps[1], NW_corner_gps
    [0], NW_corner_gps[1])

SW_corner_gps = gps.gps_newpos(SW_corner_gps[0],
    SW_corner_gps[1], SW_bring_in_direction,
    bring_in_distance)

NW_corner_gps = gps.gps_newpos(NW_corner_gps[0],
    NW_corner_gps[1], NW_bring_in_direction,
    bring_in_distance)

NE_corner_gps = gps.gps_newpos(NE_corner_gps[0],
    NE_corner_gps[1], NE_bring_in_direction,
    bring_in_distance)

SE_corner_gps = gps.gps_newpos(SE_corner_gps[0],
    SE_corner_gps[1], SE_bring_in_direction,
    bring_in_distance)

cell_corners = [SW_corner_gps, NW_corner_gps,
    NE_corner_gps, SE_corner_gps]

closest_distance = 1e15

for corner in cell_corners:
    distance = gps.gps_distance(last_wp_loc[0],

```

```

        last_wp_loc[1], corner[0], corner[1])

    if distance < closest_distance:
        closest_distance = distance

    self._task_transit_time = closest_distance/spd
    return self._task_transit_time

def getWaypointCartLocations(self):
    locations = [ ]
    for waypoint in self._waypoints:
        locations.append(waypoint.getCartesianLocation()
            )
    return locations

def getWaypointLatLonLocations(self):
    locations = [ ]
    for waypoint in self._waypoints:
        locations.append(waypoint.getLatLonLocation())
    return locations

def lay_parallel_tracks(self, bottom_left, bottom_right,
    top_left, top_right):
    bottom_list = gps.gps_divide_two_points(bottom_left
        [0],bottom_left[1],\
            bottom_right
            [0],
            bottom_right
            [1],\
            self.
            _sensor_sweep
            )
    top_list = gps.gps_divide_two_points(top_left[0],

```

```

        top_left[1],\
                                                    top_right[0],
                                                    top_right
                                                    [1],\
        self._sensor_sweep
        )

    return (bottom_list , top_list)

def set_task_wps(self , uav_location):
    cell_corners = self.getCorners()
    speed = self._controller.ac_limits.nom_fwd_speed
    altitude = self._controller.ap_wpt.z
    #initialize again here so that this object's
    trajectory is cleared
    self._waypoints = [ ]
    self._wp_sequence = paths.WaypointSequencer()
    self._trajectory = paths.Trajectory()

    #THESE ARE CARTESIAN
    SW_corner_cart = cell_corners[0][1]
    NW_corner_cart = cell_corners[1][1]
    NE_corner_cart = cell_corners[2][1]
    SE_corner_cart = cell_corners[3][1]

    SW_corner_gps = gps.gps_offset( Cell.AREA_SW_LAT,
        Cell.AREA_SW_LON, \
                                                    SW_corner_cart
                                                    [1],
                                                    SW_corner_cart
                                                    [0])
    NW_corner_gps = gps.gps_offset( Cell.AREA_SW_LAT,

```

```

Cell.AREA_SW_LON, \
                                NW_corner_cart
                                [1],
                                NW_corner_cart
                                [0])
NE_corner_gps = gps.gps_offset(Cell.AREA_SW_LAT,
                                Cell.AREA_SW_LON, \
                                NE_corner_cart
                                [1],
                                NE_corner_cart
                                [0])
SE_corner_gps = gps.gps_offset(Cell.AREA_SW_LAT,
                                Cell.AREA_SW_LON, \
                                SE_corner_cart
                                [1],
                                SE_corner_cart
                                [0])

```

```

#bring corners in by 50 meters.
bring_in_distance = 50.0
SW_bring_in_direction = gps.gps_bearing(
    SW_corner_gps[0], SW_corner_gps[1], NE_corner_gps
    [0], NE_corner_gps[1])
NE_bring_in_direction = gps.gps_bearing(
    NE_corner_gps[0], NE_corner_gps[1], SW_corner_gps
    [0], SW_corner_gps[1])
NW_bring_in_direction = gps.gps_bearing(
    NW_corner_gps[0], NW_corner_gps[1], SE_corner_gps
    [0], SE_corner_gps[1])
SE_bring_in_direction = gps.gps_bearing(
    SE_corner_gps[0], SE_corner_gps[1], NW_corner_gps
    [0], NW_corner_gps[1])

```

```

SW_corner_gps = gps.gps_newpos(SW_corner_gps[0],
    SW_corner_gps[1], SW_bring_in_direction ,
    bring_in_distance)

NW_corner_gps = gps.gps_newpos(NW_corner_gps[0],
    NW_corner_gps[1], NW_bring_in_direction ,
    bring_in_distance)

NE_corner_gps = gps.gps_newpos(NE_corner_gps[0],
    NE_corner_gps[1], NE_bring_in_direction ,
    bring_in_distance)

SE_corner_gps = gps.gps_newpos(SE_corner_gps[0],
    SE_corner_gps[1], SE_bring_in_direction ,
    bring_in_distance)

uav_loc = uav_location[0], uav_location[1]

northEastCornerLoop = False

if northEastCornerLoop:
    self.addWaypoint(NE_corner_gps , altitude , speed)
    self.addWaypoint(SE_corner_gps , altitude , speed)

    self.addWaypoint(SW_corner_gps , altitude , speed)
    self.addWaypoint(NW_corner_gps , altitude , speed)
    self.lastWaypoint = NW_corner_gps
    return

bottom_track , top_track = self.lay_parallel_tracks(
    SW_corner_gps , SE_corner_gps , NW_corner_gps ,
    NE_corner_gps)

```

```

closest_distance = 1000000000000

for corner in cell_corners:
    corner_localization = corner[0]
    corner = corner[1]
    corner_lat_lon = gps.gps_offset(Cell.AREA_SW_LAT
        , Cell.AREA_SW_LON, \
                                   corner[1], corner
                                   [0])

    distance = gps.gps_distance(uav_loc[0], uav_loc
        [1], corner_lat_lon[0], corner_lat_lon[1])

    if distance < closest_distance:
        corner_name = corner_localization
        closest_distance = distance

#top
track_switch = True
track = (bottom_track , top_track)

start_point = corner_name
#start_point = "SW"

#left to right progression of laying trajectory
if start_point in ["SW", "NW"]:
    index = 0
    if start_point == "SW":
        #bottom
        track_switch = track_switch
else:

```

```

        index = -1
        if start_point == "SE":
            #bottom
            track_switch = not track_switch

track_selection = track[track_switch]
#add first waypoint
self.addWaypoint(track_selection.pop(index),
    altitude , speed)

while len(bottom_track) != 0 and len(top_track) !=
0:
    #toggle between top/bottom track.
    track_switch = not track_switch
    track_selection = track[track_switch]
    #remove two waypoints from the array
    self.addWaypoint(track_selection.pop(index),
        altitude , speed)
    self.addWaypoint(track_selection.pop(index),
        altitude , speed)

track_switch = not track_switch
track_selection = track[track_switch]
#add the final waypoint
self.lastWaypoint = track_selection[-1]
self.addWaypoint(track_selection.pop(index),
    altitude , speed)
return

def addWaypoint(self , waypoint , altitude , speed):
    self._waypoints.append(waypoint)
    self._trajectory.append(waypoint[0], waypoint[1],
        altitude , speed)

```

```

        self._wp_sequence.append( waypoint[0], waypoint[1],
                                   altitude, speed)

    def deleteWaypoints( self ):
        self._waypoints = [ ]

    def setSize( self, size ):
        self._size = size

    def setWestBound( self, w ):
        self._west_bound = w

    def setEastBound( self, e ):
        self._east_bound = e

```

A.4 waypoint_paths Source Code

*# Contains functions and classes for computing and following
a specific path*

#

Duane Davis 2018

```
import math
```

```
import arsenal_library.gps_utils as gps
```

```
import arsenal_library.math_utils as math_utils
```

```
import arsenal_msgs.msg as armsg
```

```
STD_L1_DIST = 100.0
```

```
class WaypointSequencer(object):
```

*''' Used to contain and manage an ordered sequence of waypoints
Each waypoint is maintained as an LLAS object containing the
latitude , longitude , and rel_alt , and speed (implementation of
the speed feature will rely on implementing or using classes).*

Member functions:

*restart: restarts the waypoint sequence from the beginning
length: returns the number of waypoints in the sequence
remaining_length: returns the number of remaining waypoints in the sequence
clear: removes all waypoints from the sequence
append: add a waypoint to the list
get: gets a specific waypoint from the sequence (by index)
get_current: gets the next waypoint without incrementing
get_next: gets the next waypoint (or None if at the end)
get_next_wrap: gets the next waypoint, wraps to waypoint[0] at end
distance: returns the total distance of the entire waypoint path
remaining_distance: returns the distance from the current waypoint to the path end*

Member variables:

_sequence: list of waypoint objects

```

    _current: index of the next waypoint to be retrieved
    '''

    def __init__(self):
        ''' Initializes the object
        '''

        self._sequence = []
        self._current = -1

    def restart(self):
        ''' Resets the current waypoint path to start over
        from waypoint 0
        '''

        self._current = -1

    def length(self):
        ''' Returns the number of waypoints in the sequence
        @return the number of waypoints
        '''

        return len(self._sequence)

    def remaining_length(self):
        ''' Returns the number of remaining waypoints in the
        sequence
        (i.e., the number that have not been executed yet)
        @return the number of remaining waypoints
        '''

        return len(self._sequence) - (self._current + 1)

    def percent_complete(self):

```

```

        completed = self.distance()-self.remaining_distance
        ()
    if completed <= 0.0:
        return 0.0
    total = self.distance()
    percent = round(completed/total*100)
    return percent

def clear(self):
    ''' Removes all waypoints from the sequence
    '''

    del self._sequence[:]
    self._current = -1

def append(self, lat, lon, rel_alt, speed=0.0):
    ''' Appends a new waypoint to the end of the
        sequence
    @param lat: latitude of the new waypoint
    @param lon: longitude of the new waypoint
    @param rel_alt: rel_alt of the new waypoint
    @param speed: desired speed at the sequence point
    '''

    new_wpt = armsg.LLAS()
    new_wpt.lat = lat
    new_wpt.lon = lon
    new_wpt.alt = rel_alt
    new_wpt.speed = speed
    self._sequence.append(new_wpt)

```

```

def get(self , wpt_num):
    ''' Returns a specific waypoint from the sequence (
        by index)
    Raises an IndexError if an invalid waypoint number
        is specified
    @param wpt_num: index of the requested waypoint
    '''
    return self._sequence[wpt_num]

def get_current(self):
    ''' Returns the next waypoint in the sequence
        without incrementing
    Will raise an IndexError if the waypoint sequence
        does not contain waypoints
    @return the current waypoint in the sequence (None
        if at end)
    '''
    if len(self._sequence) == 0:
        raise Exception("Cannot_get_wp_from_
            uninitialized_waypoint_sequence")
    if self._current == -1:
        raise Exception('No "current" waypoint--waypoint
            _sequence_not_started')
    if self._current >= len(self._sequence):
        return None
    return self._sequence[self._current]

def get_next(self):
    ''' Returns the next waypoint in the sequence or
        None if at the end
    Will raise an IndexError if the waypoint sequence

```

```

        does not contain waypoints
        @return: the next waypoint in the sequence (None if
            at end)
        ...

    if len(self._sequence) == 0:
        raise Exception("Cannot get wp from
            uninitialized waypoint sequence")
    if self._current >= len(self._sequence):
        return None
    self._current += 1
    wpt = self._sequence[self._current]
    return wpt

def get_next_wrap(self):
    ''' Returns the next waypoint in the sequence (wraps
        to 0 from the end)
        Will raise an IndexError if the waypoint sequence
        does not contain waypoints
        @return the next waypoint in the sequence (None if
            empty)
        ...

    if len(self._sequence) == 0:
        raise Exception("Cannot get wp from
            uninitialized waypoint sequence")
    if self._current == len(self._sequence):
        self._current = -1
    return self.get_next()

def distance(self):
    ''' Returns the total distance represented by the
        waypoint path

```

```

        @return total path length (meters) from start to
        finish
        '''
        distance = 0.0
        for segnum in range(1, len(self._sequence)):
            pt1 = self._sequence[segnum-1]
            pt2 = self._sequence[segnum]
            distance += gps.gps_distance(pt1.lat, pt1.lon,
                                         pt2.lat, pt2.lon)
        return distance

    def remaining_distance(self):
        ''' Returns the remaining distance from the current
            waypoint to the path end
            @return remaining path length (meters) from current
            waypoint to path end
            '''
        distance = 0.0
        for segnum in range(self._current, len(self._sequence)):
            pt1 = self._sequence[segnum-1]
            pt2 = self._sequence[segnum]
            distance += gps.gps_distance(pt1.lat, pt1.lon,
                                         pt2.lat, pt2.lon)
        return distance

class Trajectory(WaypointSequencer):
    ''' Used to contain and manage a planned trajectory as a
        discretized
        series of waypoints.
    '''

```

Member functions

*append: add a waypoint to the list (override of
WaypointSequencer method)*
*get_nearest_point: returns the closest on-trajectory
lat, lon, alt of an arbitrary lat/lon*
*get_intercept_point: computes an intercept point for
the UAV to "aim" track the Trajectory*
*_project_forward: projects forward on the trajectory
from a specified point*
*get_geo_point: computes a geographic lat/lon from a
Cartesian (x,y) in trajectory space*
*get_cartesian_point: computes a Cartesian point in
trajectory space from a lat/lon*

Member functions (inherited from WaypointSequencer):

*get_next: gets the next waypoint (or None if at the
end)*
*get_next_wrap: gets the next waypoint, wraps to
waypoint[0] at end*

Protected member variables

*_L1_distance: distance ahead of the trajectory to aim
for when tracking*

*Protected member variables (inherited from
WaypointSequencer):*

_sequence: list of waypoint objects
_current: index of the next waypoint to be retrieved
, , ,

```
def __init__(self, l1_dist=STD_L1_DIST):  
    ''' Initializes the object  
    @param l1_dist: along-track distance of a fly-to
```

```

        intercept point (default 50.0)
    , , ,

WaypointSequencer.__init__(self)
self._L1_distance = l1_dist
self.__cartesian_pts = []

def get_geo_point(self, x, y):
    ''' Utility function to convert a Cartesian point in
        trajectory-fixed
        space back to latitude/longitude
        @param x: x (north) position being converted
        @param y: y (east) position being converted
        @return (latitude, longitude) tuple
        , , ,

    return gps.gps_offset(self._sequence[0].lat, self._
        _sequence[0].lon, y, x)

def get_cartesian_point(self, lat, lon):
    ''' Utility function to convert an arbitrary
        latitude/longitude to
        a Cartesian point in trajectory-fixed space (X north
        )
        @param lat: latitude being converted
        @param lon: longitude being converted
        @return (x, y) tuple
        , , ,

    return gps.cartesian_offset(self._sequence[0].lat, \
        self._sequence[0].lon,
        lat, lon)

```

```

def append(self, lat, lon, rel_alt, speed=0.0):
    ''' Appends a new waypoint to the end of the
        sequence
        @param lat: latitude of the new waypoint
        @param lon: longitude of the new waypoint
        @param rel_alt: rel_alt of the new waypoint
        @param speed: desired speed at the sequence point
    '''

    super(Trajectory, self).append(lat, lon, rel_alt,
                                     speed)

    if len(self._sequence) == 1:
        self.__cartesian_pts.append((0, 0))

    else:
        self.__cartesian_pts.append(self.
                                     get_cartesian_point(lat, lon))

def get_nearest_point(self, lat, lon):
    ''' Returns the closest normal point on the
        trajectory (i.e., a point
        on the trajectory path that is closest to the
        parameter lat/lon).
        Will throw an IndexError if there are no points in
        the Trajectory
        @param lat: test latitude
        @param lon: test longitude
        @returns: a tuple containing the nearest point lat,
                  lon, (interpolated)
                  altitude, speed, segment number, and distance from
                  the test point

```

```

    , , ,

if len(self._sequence) == 0:
    raise IndexError("Cannot compute nearest point
                     to uninitialized trajectory")

if len(self._sequence) == 1:
    dist = gps.gps_distance(lat, lon, \
                             self._sequence[0].lat, \
                             self._sequence[0].lon)
    return ( self._sequence[0].lat, \
              self._sequence[0].lon, \
              self._sequence[0].alt, \
              self._sequence[0].speed, 0, dist )

# Find the closest point on each segment & overall
in Cartesian space
test_cartesian = self.get_cartesian_point(lat, lon)
seg_min = -1
d_min = 1e15
closest = None
for segment in range(0, len(self._sequence)-1):
    pt = math_utils.closest_segment_pt(self.
                                         __cartesian_pts[segment], \
                                         self.
                                         __cartesian_pts
                                         [segment
                                         +1], \
                                         test_cartesian
                                         )
    d = math.hypot(pt[0]-test_cartesian[0], pt[1]-
                   test_cartesian[1])
    if d <= d_min:
        seg_min = segment

```

```

        d_min = d
        closest = pt

    # Convert Cartesian offset from first point to lat/
    lon
    geo_pt = self.get_geo_point(closest[0], closest[1])

    # Interpolate for altitude at the point along the
    segment
    seg_length = \
        math_utils.segment_magnitude(self.
            __cartesian_pts[seg_min], \
            self.
                __cartesian_pts[
                    seg_min+1])

    pt_dist = \
        math_utils.segment_magnitude(self.
            __cartesian_pts[seg_min], closest)
    interp_alt = \
        self._sequence[seg_min].alt + \
        ((self._sequence[seg_min+1].alt - self._sequence
            [seg_min].alt) * \
            (pt_dist / seg_length))
    interp_spd = \
        self._sequence[seg_min].speed + \
        ((self._sequence[seg_min+1].speed - self.
            _sequence[seg_min].speed) * \
            (pt_dist / seg_length))
    return ( geo_pt[0], geo_pt[1], interp_alt ,
        interp_spd , seg_min , d_min )

def get_intercept_point(self , pose):

```

```

''' Computes an on-trajectory intercept point for
    the UAV to "aim" for
    in order to track along the Trajectory. The point
    will be computed
    from an arseml_msgs/SwarmVehicleState object. If
    the UAV is more than
    Ll distance from the Trajectory, the "nearest point"
    will be used. If
    the it is closer, the computed intercept will be
    forward of the
    "nearest" point by the Ll distance minus the current
    distance from the
    Trajectory. If the intercept would extend beyond
    the last point of the
    Trajectory, the last point will be utilized.
    @param pose: vehicle pose from which to derive the
    intercept point
    @return a tuple containing the nearest point lat ,
    lon , (interpolated)
    altitude , (interpolated) speed , the segment number
    of the intercept ,
    a Boolean flag indicating reaching the Trajectory ,
    and the total
    remaining trajectory distance (i.e., pose-to-
    intercept+intccept-to-end)
    , , ,

    int_lat , int_lon , interp_alt , interp_spd , int_seg ,
    at_end = \
        self.__get_intercept_point(pose)
    rem_dist = gps.gps_distance(pose.position.lat , pose.
    position.lon , \
                                int_lat , int_lon)

    if at_end:

```

```

        return (int_lat , int_lon , interp_alt , interp_spd
                , \
                    int_seg , at_end , rem_dist)
current_segment = int_seg
start_pt = (int_lat , int_lon)
while current_segment < (len(self._sequence) - 1):
    end_pt = (self._sequence[current_segment+1].lat ,
              \
                  self._sequence[current_segment+1].lon)
    rem_dist += gps.gps_distance(start_pt[0] ,
                                start_pt[1] , \
                                    end_pt[0] , end_pt
                                    [1])

    start_pt = end_pt
    current_segment += 1
return (int_lat , int_lon , interp_alt , interp_spd , \
        int_seg , at_end , rem_dist)

```

```

def __get_intercept_point(self , pose):
    ''' Private method that computes an on-trajectory
        intercept point for
        the UAV to "aim" for in order to track along the
        Trajectory. The point
        will be computed from an arsenal_msgs/
        SwarmVehicleState object. If the
        UAV is more than L1 distance from the Trajectory ,
        the "nearest point"
        will be used. If the it is closer , the computed
        intercept will be
        forward of the "nearest" point by the L1 distance
        minus the current
        distance from the Trajectory. If the intercept

```

```

        would extend beyond the
last point of the Trajectory, the last point will be
        utilized.
@param pose: vehicle pose from which to derive the
        intercept point
@return a tuple containing the nearest point lat,
        lon, (interpolated)
        altitude, (interpolated) speed, the segment number
        of the intercept,
and a Boolean flag indicating reaching the
        Trajectory
    """
    nearest = self.get_nearest_point(pose.position.lat,
        pose.position.lon)

    # More than L1 dist away, drive straight to it for
    now
    if (self._L1_distance <= nearest[5]):
        return (nearest[0], nearest[1], nearest[2],
            nearest[3], 0, False)

    # Project along the Trajectory to find the intercept
    point
    return self._project_forward(self._L1_distance -
        nearest[4], \
                                nearest[0], nearest[1],
                                nearest[4])

def _project_forward(self, proj_dist, lat, lon, seg_id):
    """ Computes a point on the trajectory that is a
        specified distance
        forward from a starting point (on the trajectory)

```

```

    @param proj_dist: distance forward from the start to
        project
    @param lat: latitude of the starting point (must be
        on the Trajectory)
    @param lon: longitude of the starting point (must be
        on the Trajectory)
    @param seg_id: ID of the Trajectory segment
        containing the starting point
    @return a tuple containing the projected on-
        trajectory point, the segment
        number of the projected point, a Boolean flag
        indicating reaching the end
    ...

    # Trivial cases: uninitialized or only one point in
        the Trajectory
    num_segments = len(self._sequence) - 1
    if (num_segments < 0):
        raise Exception("Cannot compute projected point
            to uninitialized trajectory")
    if (num_segments == 0):
        return ( self._sequence[0].lat , \
            self._sequence[0].lon , \
            self._sequence[0].alt , \
            self._sequence[0].speed , 0, True )

    # Use Cartesian coordinates to find the projected
        point
    int_pt = self.get_cartesian_point(lat , lon)
    interp_ratio = 0.0 # Used for projection and
        altitude interpolations
    at_end = False

    while (proj_dist > 0.0):

```

```

seg_pt_1 = self.__cartesian_pts[seg_id]
seg_pt_2 = self.__cartesian_pts[seg_id+1]
seg_brg = math.atan2(seg_pt_2[1] - seg_pt_1[1],
    \
                        seg_pt_2[0] - seg_pt_1[0] )
seg_len = math.hypot(seg_pt_2[0] - seg_pt_1[0],
    \
                        seg_pt_2[1] - seg_pt_1[1])
seg_offset = math.hypot(seg_pt_1[0] - int_pt[0],
    \
                        seg_pt_1[1] - int_pt[1])
seg_dist = seg_offset + proj_dist
interp_ratio = seg_dist / seg_len
if (seg_len >= seg_dist):
    int_pt = ( seg_pt_1[0] + math.cos(seg_brg) *
                seg_dist, \
                seg_pt_1[1] + math.sin(seg_brg) *
                seg_dist )
    proj_dist = -1.0 # Point found
elif (seg_id == (num_segments-1)):
    int_pt = seg_pt_2
    interp_ratio = 1.0
    proj_dist = -1.0 # Reached end of
                    Trajectory path
    at_end = True
else:
    int_pt = seg_pt_2
    seg_id += 1 # Wrap projection onto next
                segment
    proj_dist -= (seg_len - seg_offset)

# Convert Cartesian pt to lat/lon and interpolate
for altitude

```

```

seg_alt = self._sequence[seg_id].alt
interp_alt = \
    self._sequence[seg_id].alt + \
    ((self._sequence[seg_id+1].alt - self._sequence[
        seg_id].alt) * \
        interp_ratio)
interp_spd = \
    self._sequence[seg_id].speed + \
    ((self._sequence[seg_id+1].speed - self.
        _sequence[seg_id].speed) * \
        interp_ratio)
geo_pt = self.get_geo_point(int_pt[0], int_pt[1])
return ( geo_pt[0], geo_pt[1], interp_alt ,
        interp_spd , seg_id , at_end )

```

A.5 independent_transits Source Code

```

# Contains functions and classes for independently directing
a UAV to a
# destination (i.e., without regard to where other swarm
UAVs are).
#
# Duane Davis 2018

# Standard Python imports
import math

#ARSENL imports
import arsenl_data.enumerations as enums
import arsenl_library.gps_utils as gps
import arsenl_library.math_utils as math_utils
import arsenl_control.swarm_control as ctrlr
import arsenl_behavior_tools.waypoint_paths as paths

```

```

class DirectDriver(object):
    ''' Used to drive the UAV directly to a specific
        location

    Member variables:
        _controller: SwarmController object to which this
                     object belongs
        _latitude: latitude to which the vehicle is to be
                     driven
        _longitude: longitude to which the vehicle is to be
                     driven
        _rel_alt: desired altitude (meters above launch)
        _speed: speed (m/s) at which the vehicle is to transit

    Member functions:
        get_waypoint: returns the waypoint object for the
                     desired location
        reset: resets the parameters (latitude longitude ,
                     altitude , and speed)
        distance: returns distance from current location to
                     commanded location
    '''

    def __init__(self , controller , latitude , longitude ,
        altitude , speed=0.0):
        ''' Initializes the object
        @param controller: SwarmController object to which
                           this object belongs
        @param latitude: latitude to which the vehicle is to
                           be driven
        @param longitude: longitude to which the vehicle is
    '''

```

```

        to be driven
    @param altitude: desired altitude (meters above
        launch)
    @param speed: speed (m/s) at which the vehicle is to
        transit
    '''
    if not isinstance(controller, ctrl.SwarmController):
        raise TypeError("Controller must be of type \
            SwarmController")
    self._controller = controller
    self._latitude = latitude
    self._longitude = longitude
    self._rel_alt = altitude
    if speed < 1.0:
        self._speed = self._controller.ac_limits.\
            nom_fwd_speed
    elif speed < self._controller.ac_limits.\
        min_fwd_speed:
        self._speed = self._controller.ac_limits.\
            min_fwd_speed
    elif speed > self._controller.ac_limits.\
        max_fwd_speed:
        self._speed = self._controller.ac_limits.\
            max_fwd_speed
    else:
        self._speed = speed

def reset(self, latitude, longitude, altitude=0, speed
=0):
    ''' Resets the DirectDriver's parameters (
        destination, latitude
        longitude, altitude, and speed). If the altitude

```

and/or speed
parameters are "0", then the previous value will be
retained.
If the speed parameter is less than 1.0, the nominal
forward speed
will be used. If the speed parameter is less than
the minimum
speed or greater than the maximum speed, the minimum
or maximum
speed will be used respectively.
@*param* *latitude*: *updated destination latitude*
@*param* *longitude*: *updated destination longitude*
@*param* *altitude*: *desired altitude (meters above*
launch)
@*param* *speed*: *speed (m/s) at which the vehicle is to*
transit
,,,
self._latitude = latitude
self._longitude = longitude
if altitude != 0:
 self._rel_alt = altitude
if speed == 0:
 pass # *Retain previously ordered speed*
elif speed < 1.0:
 self._speed = self._controller.ac_limits.
 nom_fwd_speed
elif speed < self._controller.ac_limits.
 min_fwd_speed:
 self._speed = self._controller.ac_limits.
 min_fwd_speed
elif speed > self._controller.ac_limits.
 max_fwd_speed:
 self._speed = self._controller.ac_limits.

```

        max_fwd_speed
    else:
        self._speed = speed

def get_waypoint(self):
    ''' Returns a waypoint object for the desired
        location
    @return arseml_msgs/LLAS directing the vehicle to
        the desired location
    '''
    self._controller.spd_wp_cmd_msg.lat = self._latitude
    self._controller.spd_wp_cmd_msg.lon = self._longitude
    self._controller.spd_wp_cmd_msg.alt = self._rel_alt
    self._controller.spd_wp_cmd_msg.speed = self._speed
    return self._controller.spd_wp_cmd_msg

def distance(self):
    ''' Returns the distance from the UAV's current
        location to the
        commanded location
    @return distance (m) from the UAV to the objective
        location
    '''
    own_posit = self._controller.get_own_state().state.pose.pose.position
    return gps.gps_distance(own_posit.lat, own_posit.lon, \
                             self._latitude, self._longitude)

```

```
class DriveToDirectDriver(DirectDriver):
    ''' Used to drive the UAV to a specific location and fly
        all the way to
        the destination point (to include flying through it and
        back as reqd).
```

Member variables (inherited from DirectDriver):

```
    _controller: SwarmController object to which this
                  object belongs
    _latitude: latitude to which the vehicle is to be
               driven
    _longitude: longitude to which the vehicle is to be
                driven
    _rel_alt: desired altitude (meters above launch)
    _speed: speed (m/s) at which the vehicle is to transit
```

Member functions:

```
    get_waypoint: returns the waypoint object for the
                  location (override)
```

Member functions (inherited from DirectDriver):

```
    reset: resets the parameters (latitude longitude ,
                                   altitude , and speed)
    distance: returns distance from current location to
              commanded location
    , , ,
```

```
FLY_TO_DIST = 250.0
```

```
def get_waypoint(self):
    ''' Returns a waypoint object for the desired
        location
```

```

        @return arseml_msgs/LLAS directing the vehicle to
            the desired location
        '''
        own_posit = self._controller.get_own_state().state.
            pose.pose.position
        bearing = gps.gps_bearing(own_posit.lat , own_posit.
            lon , \
                                self._latitude , self.
                                _longitude)
        new_pos = gps.gps_newpos(own_posit.lat , own_posit.
            lon , \
                                bearing ,
                                DriveToDirectDriver.
                                FLY_TO_DIST)
        self._controller.spd_wp_cmd_msg.lat = new_pos[0]
        self._controller.spd_wp_cmd_msg.lon = new_pos[1]
        self._controller.spd_wp_cmd_msg.alt = self._rel_alt
        self._controller.spd_wp_cmd_msg.speed = self._speed
        return self._controller.spd_wp_cmd_msg

```

```

class TrackToDriver(DriveToDirectDriver):

```

```

    ''' Used to drive the UAV to a specific location via a
        specific track
        (path) flying all the way to the destination point (to
            include flying
            through it and back as reqd).

```

Member variables:

```

    _start_latitude: beginning latitude of the intended
        track
    _start_longitude: beginning longitude of the intended
        track

```

_track_length: length of the track being followed
_track_course: vehicle course when following track

Member variables (inherited from DirectDriver):

*_controller: SwarmController object to which this
object belongs*
*_latitude: latitude to which the vehicle is to be
driven*
*_longitude: longitude to which the vehicle is to be
driven*
_rel_alt: desired altitude (meters above launch)
_speed: speed (m/s) at which the vehicle is to transit

Member functions:

*get_waypoint: returns the waypoint object for the
location (override)*
reset_start: resets the starting point of the track

Member functions (inherited from DirectDriver):

*reset: resets the parameters (latitude longitude ,
altitude , and speed)*
*distance: returns distance from current location to
commanded location*
, , ,

*CORRECT_DIST = 100.0 # Distance ahead of UAV for track
intercept*

```
def __init__(self , controller , start_lat , start_lon ,  
            end_lat , end_lon , \  
                altitude , speed=0.0):  
    ''' Initializes the object  
    @param controller: SwarmController object to which
```

```

        this object belongs
        @param start_lat: starting latitude of intended
            vehicle track
        @param start_lon: starting longitude of intended
            vehicle track
        @param end_lat: latitude to which the vehicle is to
            be driven
        @param end_lon: longitude to which the vehicle is to
            be driven
        @param altitude: desired altitude (meters above
            launch)
        @param speed: speed (m/s) at which the vehicle is to
            transit
        ...
super(TrackToDriver, self).__init__(controller,
            end_lat, end_lon, \
                                   altitude, speed)

        self._start_latitude = start_lat
        self._start_longitude = start_lon
        self._track_length = \
            gps.gps_distance(self._start_latitude, self.
                _start_longitude, \
                    self._latitude, self._longitude
                )
        self._track_course = \
            gps.gps_bearing(self._start_latitude, self.
                _start_longitude, \
                    self._latitude, self._longitude)

def reset_start(self, latitude, longitude):
    ''' Resets the starting point of the desired vehicle
        track

```

```

        @param latitude: latitude of track starting point
        @param longitude: longitude of track startingpoint
        '''
        self._start_latitude = latitude
        self._start_longitude = longitude
        self._track_length = \
            gps.gps_distance(self._start_latitude , self._
                            _start_longitude , \
                                self._latitude , self._longitude
                            )
        self._track_course = \
            gps.gps_bearing(self._start_latitude , self._
                            _start_longitude , \
                                self._latitude , self._longitude)

    def get_waypoint(self):
        ''' Returns a waypoint object for the desired
            location
        @return arseml_msgs/LLAS directing the vehicle to
            the desired location
        '''
        # Determine the along-track travel & cross-track
            error
        uav_posit = self._controller.get_own_state().state.
            pose.pose.position
        to_uav_brg = \
            gps.gps_bearing(self._start_latitude , self._
                            _start_longitude , \
                                uav_posit.lat , uav_posit.lon)
        to_uav_dst = \
            gps.gps_distance(self._start_latitude , self._
                            _start_longitude , \

```

```

        uav_posit.lat , uav_posit.lon)
course_error = math_utils.normalize_pi(to_uav_brg -
    self._track_course)
to_end_dst = \
    gps.gps_distance(uav_posit.lat , uav_posit.lon , \
        self._latitude , self._longitude
    )
along_track_tv1 = to_uav_dst * math.cos(course_error
    )
cross_track_err = to_uav_dst * math.sin(course_error
    )

# If we're already past the waypoint or closer than
# the correction distance , just fly straight there
if (to_end_dst <= TrackToDriver.CORRECT_DIST) or \
    (along_track_tv1 >= (self._track_length - \
        TrackToDriver.CORRECT_DIST))
    :
    return super(TrackToDriver , self).get_waypoint()

# Find an intercept point and fly towards it
correction_tgt_dist = along_track_tv1 + \
    max(0.0 , TrackToDriver.
        CORRECT_DIST - \
            abs(cross_track_err))

intercept = \
    gps.gps_newpos(self._start_latitude , self.
        _start_longitude , \
            self._track_course ,
            correction_tgt_dist)
fly_hdg = gps.gps_bearing(uav_posit.lat , uav_posit.
    lon , \
        intercept[0] , intercept

```

```

[1])
wp_posit = gps.gps_newpos(uav_posit.lat , uav_posit.
    lon , \
        fly_hdg ,
        DriveToDirectDriver.
        FLY_TO_DIST)
self._controller.spd_wp_cmd_msg.lat = wp_posit[0]
self._controller.spd_wp_cmd_msg.lon = wp_posit[1]
self._controller.spd_wp_cmd_msg.alt = self._rel_alt
self._controller.spd_wp_cmd_msg.speed = self._speed
return self._controller.spd_wp_cmd_msg

```

```

class WaypointSequenceDriver(object):

```

```

    ''' Used to drive generate waypoint commands to transit
        through a series
    of waypoints in order. Upon reaching the end, the last
        waypoint will be
    orbited and a "complete" flag will be set to True.

```

Member variables:

```

    _controller: SwarmController object to which this
        object belongs
    _sequence: WaypointSequencer that is to be followed
    _current_driver: reference to the driver object
        currently in use
    _path_driver: driver object for transiting through the
        sequence
    _final_driver: driver object for orbiting the final
        point
    _complete: Boolean flag indicating that the sequence
        has completed
    _started: Boolean flag indicating that the sequence

```

has been started

Member functions:

reset: sets a new sequence to be followed
get_waypoint: returns a waypoint object for following
the path

is_complete: returns True if the search pattern is
complete

, , ,

#CAPTURE_DIST = 45.0

#PATH_END_CAPTURE_DIST = 85.0

CAPTURE_DIST = 60.0

PATH_END_CAPTURE_DIST = 120.0

def __init__(self, controller):

''' *Class initializer for class variables*

@param controller: SwarmController object to which
this object belongs

, , ,

if not isinstance(controller, ctrl.SwarmController):

raise TypeError("controller must be of type
SwarmController")

self._controller = controller

self._sequence = None

self._current_driver = None

self._path_driver = None

self._final_driver = None

self._complete = False

self._started = False

def is_complete(self):

```

''' Return True if the search pattern has been
    completed
    @return Boolean indicating completion of the search
        path
    '''
    return self._complete

def reset(self, sequence):
    ''' Resets the waypoint sequence that is to be
        followed
    @param sequence: waypoint sequence to be followed
    '''
    if not isinstance(sequence, paths.WaypointSequencer)
        :
        raise TypeError("waypoint_sequencer must be of
            type_WaypointSequencer")
    if not sequence.length() > 0:
        raise TypeError("attempt to reset to empty
            WaypointSequencer")
    self._sequence = sequence
    self._sequence.restart()
    self._complete = False
    self._started = False
    first_wp = sequence.get_next()
    last_wp = sequence.get(-1)
    if not self._path_driver or not self._final_driver:
        self._path_driver = \
            DriveToDirectDriver(self._controller, \
                                first_wp.lat, first_wp.
                                    lon, \
                                first_wp.alt, first_wp.
                                    speed)

```

```

        self._final_driver = \
            DirectDriver(self._controller, last_wp.lat,
                        last_wp.lon, \
                            last_wp.alt, last_wp.speed)
    else:
        self._path_driver.reset(first_wp.lat, first_wp.
                                lon, \
                                    first_wp.alt, first_wp.
                                        speed)
        self._final_driver.reset(last_wp.lat, last_wp.
                                lon, \
                                    last_wp.alt, last_wp.
                                        speed)
    self._controller.log_info("Driver commencing first \
        waypoint: \%(first_wp.lat, first_wp.
                                lon)) \
        %(first_wp.alt, first_wp.
                                speed)")
    self._current_driver = self._path_driver

def get_waypoint(self):
    ''' Returns a waypoint to drive the vehicle through
        the sequence
        and determines when the sequence has been completed.
        @return waypoint command (LLAS message)
    '''
    if not self._current_driver:
        raise TypeError("cannot get waypoint from \
            uninitialized waypoint sequence driver")

    if not self._started:
        self._started = True

```

```

        this_wp = self._sequence.get_current()
        self._controller.log_info("driver commencing \
            first waypoint: \%(this_wp.lat , this_wp
                                .lon))

remaining = self._sequence.remaining_length()
distance = self._current_driver.distance()
if remaining == 0 and not self._complete and \
    distance <= WaypointSequenceDriver.
        PATH_END_CAPTURE_DIST:
        self._complete = True
        self._controller.log_info("driver arrived at \
            final waypoint")

elif remaining == 1 and distance <=
    WaypointSequenceDriver.CAPTURE_DIST:
        last_wp = self._sequence.get_next()
        self._current_driver = self._final_driver
        self._controller.log_info("driver transitioning \
            to final waypoint: \%(last_wp.lat , last_wp
                                .lon))

elif remaining > 0 and \
    distance <= WaypointSequenceDriver.CAPTURE_DIST
        :
        next_wp = self._sequence.get_next()
        self._current_driver.reset(next_wp.lat , next_wp.
            lon , \
                                next_wp.alt , next_wp.
                                speed)
        self._controller.log_info("driver transitioning \

```

```

        to_next_waypoint: "%f/%f" \
                                %(next_wp.lat , next_wp
                                .lon))

    return self._current_driver.get_waypoint()

class TrackToWaypointSequenceDriver(WaypointSequenceDriver):
    ''' Used to drive generate waypoint commands to transit
        through a series
        of waypoints in order. Upon reaching the end, the last
        waypoint will be
        orbited and a "complete" flag will be set to True.

    Member variables:
        _path_start_lat: latitude from which the waypoint
            transit is starting
        _path_start_lon: longitude from which the waypoint
            transit is starting

    Inherited member variables (WaypointSequenceDriver):
        _controller: SwarmController object to which this
            object belongs
        _sequence: WaypointSequencer that is to be followed
        _current_driver: reference to the driver object
            currently in use
        _path_driver: driver object for transiting through the
            sequence
        _final_driver: driver object for orbiting the final
            point
        _complete: Boolean flag indicating that the sequence
            has completed
        _started: Boolean flag indicating that the sequence

```

has been started

Member functions:

reset: sets a new sequence to be followed (override)

*reset_start: Resets the starting position of the
waypoint path*

*get_waypoint: returns a waypoint object for following
the path (override)*

Inherited member functions (WaypointSequenceDriver):

*is_complete: returns True if the search pattern is
complete*

, , ,

```
def __init__(self, controller):  
    ''' Class initializer for class variables  
    @param controller: SwarmController object to which  
        this object belongs  
    , , ,  
  
    super(TrackToWaypointSequenceDriver, self).__init__(  
        controller)  
    posit = self._controller.get_own_state().state.pose.  
        pose.position  
    self._path_start_lat = posit.lat  
    self._path_start_lon = posit.lon
```

```
def reset_start(self, start_lat, start_lon):  
    ''' Resets the starting position of the waypoint  
        path  
    @param start_lat: new starting latitude  
    @param start_lon: new starting longitude
```

```

    , , ,

    self._path_start_lat = start_lat
    self._path_start_lon = start_lon
    if self._sequence:
        self.reset(self._sequence)

def reset(self, sequence):
    ''' Resets the waypoint sequence that is to be
        followed
        (parent class override)
        @param sequence: waypoint sequence to be followed
    '''
    if not isinstance(sequence, paths.WaypointSequencer)
        :
        raise TypeError("waypoint_sequencer_must_be_of_
            type_WaypointSequencer")
    if not sequence.length() > 0:
        raise TypeError("attempt_to_reset_to_empty_
            WaypointSequencer")
    self._sequence = sequence
    self._sequence.restart()
    self._complete = False
    self._started = False
    first_wp = sequence.get_next()
    last_wp = sequence.get(-1)
    posit = self._controller.get_own_state().state.pose.
        pose.position
    self._path_start_lat = posit.lat
    self._path_start_lon = posit.lon
    if not self._path_driver or not self._final_driver:
        self._path_driver = \
            TrackToDriver(self._controller, \

```

```

        self._path_start_lat , self.
            _path_start_lon , \
            first_wp.lat , first_wp.lon , \
            first_wp.alt , first_wp.speed)
self._path_driver.reset_start(self.
    _path_start_lat , \
                                self.
                                    _path_start_lon
                                )
self._final_driver = \
    DirectDriver(self._controller , last_wp.lat ,
        last_wp.lon , \
            last_wp.alt , last_wp.speed)
else :
    self._path_driver.reset(first_wp.lat , first_wp.
        lon , \
                                first_wp.alt , first_wp.
                                    speed)
    self._path_driver.reset_start(self.
        _path_start_lat , \
                                self.
                                    _path_start_lon
                                )
    self._final_driver.reset(last_wp.lat , last_wp.
        lon , \
                                last_wp.alt , last_wp.
                                    speed)
self._current_driver = self._path_driver

```

```

def get_waypoint(self):
    ''' Returns a waypoint to drive the vehicle through
        the sequence

```

```

and determines when the sequence has been completed.
(parent class override)
@return waypoint command (LLAS message)
    , , ,

if not self._current_driver:
    raise TypeError("cannot_get_waypoint_from_
        uninitialized_waypoint_sequence_driver")

if not self._started:
    self._started = True
    this_wp = self._sequence.get_current()
    self._controller.log_info("driver_commencing_
        first_track:_%f/%f_to_%f/%f"\
                                %(self._path_start_lat
                                    , self.
                                        _path_start_lon , \
                                        this_wp.lat , this_wp
                                            .lon))

remaining = self._sequence.remaining_length()
distance = self._current_driver.distance()
if remaining == 0 and not self._complete and \
    distance <= WaypointSequenceDriver.
        PATH_END_CAPTURE_DIST:
    self._complete = True
    self._controller.log_info("driver_arrived_at_
        final_path_waypoint")

elif remaining == 1 and distance <=
    WaypointSequenceDriver.CAPTURE_DIST:
    last_wp = self._sequence.get_next()
    self._current_driver = self._final_driver
    self._controller.log_info("driver_transitioning_

```

```

        to_final_path_waypoint: "%f/%f" \
                                   %(last_wp.lat , last_wp
                                   .lon))

    elif remaining > 0 and \
        distance <= WaypointSequenceDriver.CAPTURE_DIST
        :
        this_wp = self._sequence.get_current()
        next_wp = self._sequence.get_next()
        self._current_driver.reset(next_wp.lat , next_wp.
        lon , \
                                   next_wp.alt , next_wp.
                                   speed)
        self._current_driver.reset_start(this_wp.lat ,
        this_wp.lon)
        self._controller.log_info("driver_transitioning \
        to_next_track: "%f/%f to "%f/%f" \
                                   %(this_wp.lat , this_wp
                                   .lon , next_wp.lat ,
                                   next_wp.lon))

    #bc: calls an instance of direct_to_driver.
    getwaypoint which returns a spd_wpt_cmd_msg
    return self._current_driver.get_waypoint()

```

```

class TrajectoryFollower(object):

```

```

    ''' Used to generate waypoint commands to follow a
        specific trajectory

```

```

    Member variables:

```

```

        _controller: SwarmController object to which this
        object belongs

```

_trajectory: paths.Trajectory that is being followed
_remaining_dist: remaining travel distance along the trajectory

Member functions:

compute_waypoint: computes a trajectory-following waypoint

, , ,

```
def __init__(self, controller, trajectory):
    ''' Initializes the object
    @param controller: SwarmController object that this object supports
    @param trajectory: paths.Trajectory that is to be followed
    , , ,

    if not isinstance(controller, ctrl.SwarmController):
        raise TypeError("Controller must be of type SwarmController")
    if not isinstance(trajectory, paths.Trajectory):
        raise TypeError("Trajectory must be of type Trajectory")
    self._controller = controller
    self._trajectory = trajectory
    self._remaining_dist = None

def get_intercept(self):
    ''' Generates an intercept waypoint
    @return the intercept waypoint
    , , ,

    pose = self._controller.get_own_state().state.pose.pose
```

```

        intercept = self._trajectory.get_intercept_point(
            pose)
        self._controller.spd_wp_cmd_msg.lat = intercept[0]
        self._controller.spd_wp_cmd_msg.lon = intercept[1]
        self._controller.spd_wp_cmd_msg.alt = intercept[2]
        self._remaining_dist = intercept[6]

    if intercept[3] < 1.0:
        self._controller.spd_wp_cmd_msg.speed = \
            self._controller.ac_limits.nom_fwd_speed
    elif intercept[3] < self._controller.ac_limits.
        min_fwd_speed:
        self._controller.spd_wp_cmd_msg.speed = \
            self._controller.ac_limits.min_fwd_speed
    elif intercept[3] > self._controller.ac_limits.
        max_fwd_speed:
        self._controller.spd_wp_cmd_msg.speed = \
            self._controller.ac_limits.max_fwd_speed
    else:
        self._controller.spd_wp_cmd_msg.speed =
            intercept[3]
    return self._controller.spd_wp_cmd_msg

def get_remaining_distance(self):
    ''' Returns the remaining transit distance along the
        trajectory
    '''
    return self._remaining_dist

```

List of References

- [1] J. Gertler, “U.S. unmanned aerial systems.” Library of Congress Washington DC Congressional Research Service, 2012.
- [2] G. Cai, K.-Y. Lum, B. M. Chen, and T. H. Lee, “A brief overview on miniature fixed-wing unmanned aerial vehicles,” in *IEEE ICCA 2010*. IEEE, 2010, pp. 285–290.
- [3] G. Beni, “From swarm intelligence to swarm robotics,” in *International Workshop on Swarm Robotics*. Springer, 2004, pp. 1–9.
- [4] E. Şahin, “Swarm robotics: From sources of inspiration to domains of application,” in *International workshop on swarm robotics*. Springer, 2004, pp. 10–20.
- [5] L. E. Parker, “Heterogeneous multi-robot cooperation,” Massachusetts Institute of Technology Cambridge Artificial Intelligence Lab, Tech. Rep., 1994.
- [6] T. Balch, “Hierarchic social entropy: An information theoretic measure of robot group diversity,” *Autonomous robots*, vol. 8, no. 3, pp. 209–238, 2000.
- [7] L. E. Parker, “Multiple mobile robot systems,” *Springer Handbook of Robotics*, pp. 921–941, 2008.
- [8] O. Ozdemir, “Evaluation and comparison of freedom class LCS and other frigates/corvettes against small boat, FPB and submarine threats in confined waters,” Master’s thesis, Naval Postgraduate School, Monterey, CA, 2009.
- [9] M. Teeuw, “Introducing the remote sensing of hazardous terrain,” *Geological Society, London, Special Publications*, vol. 283, no. 1, pp. 1–3, 2007.
- [10] M. S. Hopchak, “Autonomous decision in multi-robot systems,” Master’s thesis, Naval Postgraduate School, Monterey, CA, 2018.
- [11] D. Bertsekas, “A distributed algorithm for the assignment problem,” *Laboratory for Information and Decision Systems Working Paper, Massachusetts Institute of Technology, Cambridge, MA*, 1979.
- [12] R. E. Burkard, “Selected topics on assignment problems,” *Discrete Applied Mathematics*, vol. 123, no. 1-3, pp. 257–302, 2002.
- [13] D. P. Bertsekas, *Linear network optimization: algorithms and codes*. MIT Press, 1991.

- [14] R. Jonker and T. Volgenant, "Improving the hungarian assignment algorithm," *Operations Research Letters*, vol. 5, no. 4, pp. 171–175, 1986.
- [15] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval research logistics quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.
- [16] G. T. Ross and R. M. Soland, "A branch and bound algorithm for the generalized assignment problem," *Mathematical programming*, vol. 8, no. 1, pp. 91–103, 1975.
- [17] R. Cohen, L. Katzir, and D. Raz, "An efficient approximation for the generalized assignment problem," *Information Processing Letters*, vol. 100, no. 4, pp. 162–166, 2006.
- [18] M. B. Dias and A. Stentz, "A free market architecture for distributed control of a multirobot system," in *6th International Conference on Intelligent Autonomous Systems (IAS-6)*, 2000, pp. 115–122.
- [19] L. Özbakir, A. Baykasoğlu, and P. Tapkan, "Bees algorithm for generalized assignment problem," *Applied Mathematics and Computation*, vol. 215, no. 11, pp. 3782 – 3795, 2010. Available: <http://www.sciencedirect.com/science/article/pii/S0096300309010078>
- [20] B. Schwartz, "A computational analysis of the auction algorithm," *European Journal of Operational Research*, vol. 74, no. 1, pp. 161 – 169, 1994. Available: <http://www.sciencedirect.com/science/article/pii/0377221794902143>
- [21] M. B. Dias, R. Zlot, N. Kalra, and A. Stentz, "Market-based multirobot coordination: A survey and analysis," *Proceedings of the IEEE*, vol. 94, no. 7, pp. 1257–1270, 2006.
- [22] R. Simmons, D. Apfelbaum, D. Fox, R. P. Goldman, K. Z. Haigh, D. J. Musliner, M. Pelican, and S. Thrun, "Coordinated deployment of multiple, heterogeneous robots," in *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)(Cat. No. 00CH37113)*. IEEE, 2000, vol. 3, pp. 2254–2260.
- [23] R. Simmons, S. Singh, D. Hershberger, J. Ramos, and T. Smith, "First results in the coordination of heterogeneous robots for large-scale assembly," in *Experimental Robotics VII*. Springer, 2001, pp. 323–332.
- [24] D. Bertsekas, "The auction algorithm: A distributed relaxation method for the assignment problem," *Annals of Operations Research*, vol. 14, no. 1, pp. 105–123, 1988, doi: 10.1007/BF02186476.

- [25] D. Bertsekas, “Auction algorithms for network flow problems: A tutorial introduction,” *Computational Optimization and Applications*, vol. 1, no. 1, pp. 7–66, 1992, doi: 10.1007/BF00247653.
- [26] D. P. Bertsekas, “Auction algorithms,” *Encyclopedia of optimization*, pp. 128–132, 2009.
- [27] P. Cramton, Y. Shoham, and R. Steinberg, *Combinatorial Auctions*. The MIT Press, 2010.
- [28] T. H. Chung, M. R. Clement, M. A. Day, K. D. Jones, D. Davis, and M. Jones, “Live-fly, large-scale field experimentation for large numbers of fixed-wing uavs,” in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 1255–1262.
- [29] M. A. Day, M. R. Clement, J. D. Russo, D. Davis, and T. H. Chung, “Multi-uav software systems and simulation architecture,” in *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*, June 2015, pp. 426–435.
- [30] D. T. Davis, T. H. Chung, M. R. Clement, and M. A. Day, “Consensus-based data sharing for large-scale aerial swarm coordination in lossy communications environments,” in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct 2016, pp. 3801–3808.
- [31] D. Horner and G. Xie, “Data-driven acoustic communication modeling for undersea collaborative navigation,” in *2013 OCEANS-San Diego*. IEEE, 2013, pp. 1–10.

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
Ft. Belvoir, Virginia
2. Dudley Knox Library
Naval Postgraduate School
Monterey, California