NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AUTONOMOUS DECISION IN MULTI-ROBOT SYSTEMS

by

Matthew S. Hopchak

December 2018

Thesis Advisor: Duane T. Davis
Co-Advisor: Kathleen B. Giles

Approved for public release. Distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
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** December 2018	**3. REPORT TYPE AND DATES COVERED** Master's thesis	
4. TITLE AND SUBTITLE AUTONOMOUS DECISION IN MULTI-ROBOT SYSTEMS			**5. FUNDING NUMBERS**
6. AUTHOR(S) Matthew S. Hopchak			
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)

This research evaluates potential auction algorithm approaches to a multi-robot area search problem and uses the Naval Postgraduate School Advanced Robotic System Engineering Laboratory's multi-UAV system to implement, test, and evaluate selected exemplars. Ultimately, for multi-robot systems to achieve useful objectives autonomously, they need to reliably analyze objectives and assign supporting tasks to individual vehicles. The market-based approaches analyzed in this research provide an intuitive mechanism for robust realization of this capability in highly dynamic and uncertain environments. We present our implementation, AuctionSearch, evaluate its design trade-offs, and influence agent bidding strategies based on per-robot speed and endurance. We test our implementation in simulation and in live-fly experiments across three different search areas with system sizes ranging from three to 10 robots each. The future of warfare will include unmanned systems in many facets of operations and support. Furthermore, it is likely that human intervention and direct handling of autonomous systems' actions will be replaced by human supervision of autonomously developed courses of action on the battlefield. For multi-robot systems to have the capacity to develop and execute complex courses of action, they must be capable of linking complex tasks together. Our research and testing demonstrate that auction algorithms are well suited for autonomous decision.

14. SUBJECT TERMS robotics, autonomous systems, auction, ARSENL, swarm, autonomous, multi-robot systems, area search			**15. NUMBER OF PAGES** 217
			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

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

AUTONOMOUS DECISION IN MULTI-ROBOT SYSTEMS

Matthew S. Hopchak
Major, United States Army
BA, Virginia Military Institute, 2007

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2018

Approved by: Duane T. Davis
Advisor

Kathleen B. Giles
Co-Advisor

Peter J. Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This research evaluates potential auction algorithm approaches to a multi-robot area search problem and uses the Naval Postgraduate School Advanced Robotic System Engineering Laboratory's multi-UAV system to implement, test, and evaluate selected exemplars. Ultimately, for multi-robot systems to achieve useful objectives autonomously, they need to reliably analyze objectives and assign supporting tasks to individual vehicles. The market-based approaches analyzed in this research provide an intuitive mechanism for robust realization of this capability in highly dynamic and uncertain environments. We present our implementation, AuctionSearch, evaluate its design trade-offs, and influence agent bidding strategies based on per-robot speed and endurance. We test our implementation in simulation and in live-fly experiments across three different search areas with system sizes ranging from three to 10 robots each. The future of warfare will include unmanned systems in many facets of operations and support. Furthermore, it is likely that human intervention and direct handling of autonomous systems' actions will be replaced by human supervision of autonomously developed courses of action on the battlefield. For multi-robot systems to have the capacity to develop and execute complex courses of action, they must be capable of linking complex tasks together. Our research and testing demonstrate that auction algorithms are well suited for autonomous decision.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

List of Acronyms and Abbreviations

ARM Agent Resource Mapped

ARSENL Advanced Robotic Systems Engineering Laboratory

AUSM Adaptive User Selection Mechanism

BDA Battle Damage Assessment

CAP Combinatorial Auction Problem

DoD Department of Defense

ISR Intelligence, Surveillance, and Reconnaissance

MASC Mission-Based Architecture for Swarm Composability

MMKP Multi-Dimension Multiple-Choice Knapsack Problem

MRIC Multi-Robot Independent Cueing

NPS Naval Postgraduate School

SITL Software-in-the-Loop

SMR Simultaneous Multiple Round

SPP Set Packing Problem

UAV Unmanned Aerial Vehicle

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

I thank my advisors, Dr. Duane Davis and CDR Kathleen Giles, for their guidance and attentive help during the course of this research. Thank you for keeping my feet firmly on the ground, which ultimately led to me finishing in a reasonable amount of time. I also thank the entire ARSENL team for the coordination and execution of the live-flight testing that not only validated our results, but took the project out of the simulation and into the real world. Finally, I thank my wife, Ashley, for her consistent support from the very beginning of this adventure.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

1.1 Motivation

For military ground forces to be effective in their area of operation, they require an accurate view of the operational environment. Sending human scouts into the environment to assemble this comprehensive view can be unacceptably dangerous or risky if it removes too much combat power from the core unit. The United States military has robotic systems in its inventory that facilitate autonomous exploration of operational environments, but these systems still utilize a "drive by wire" solution where a human handler is responsible for decision making, maneuvering, and interpreting the results [1]. As more autonomous systems are utilized by the Department of Defense (DoD), the ability of those systems to coordinate among themselves to solve problems and make decisions could have far-reaching tactical and strategic implications.

The future of warfare will include autonomous systems in many facets of operation and support. Further, it is very likely that human intervention and direct handling of autonomous systems' actions will be replaced by human supervision of autonomously developed courses of action on the battlefield [1]–[7]. Interoperability and scalability will demand solutions for robot-to-robot coordination, cuing, and decision making among others. Towards this end, this research explores the use of market-based approaches for robot-to-robot coordination of complex behaviors. Exploration of autonomous system behaviors that maximize independent coordination will ultimately lead to combat-enhancing capabilities within the DoD today and in the future.

1.2 Research Objectives

This thesis explores the use of auction algorithms for multi-robot area search with different utility functions implemented using the Advanced Robotic Systems Engineering Laboratory (ARSENL) multi-Unmanned Aerial Vehicle (UAV) swarm as the test-bed for our implementation. Most existing solutions for multi-robot search require centralized control and likewise suffer from central points of failure. More robust and failure-tolerant solutions can be obtained using decentralized assignment using auction algorithms.

In this work we present an area search implementation called `AuctionSearch` which uses auction algorithms to generate assignments of agents to sections of a given search area. We first explore different variants of market-based assignment algorithms and then apply them to our implementation. We then observe our implementation in three different search areas with two different utility functions with multi-robot system sizes ranging from three to 10 robots each. Finally, we validate our results with live-flight testing of `AuctionSearch`.

1.3 Related Work

Autonomous coordination among robotic systems has garnered a wide range of research attention over the years as computational power and network speeds have increased. It also takes on many shapes and directions as the terms "autonomous" and "coordination" can apply to a range of independence, scale, and complexity. This thesis defines autonomous coordination as the collective determination of follow-on actions by agents free from human-handler intervention. Many advancements have been made in the multi-robot coordination arena in recent years, ranging from taxonomies of robot behaviors as in [8], control of self organized flocking techniques as in [9], [10] to large, complex robotic swarm formations such as the 50-strong ARSENL multi-UAV swarm at Naval Postgraduate School (NPS) and Harvard University's 1000 Kilobots [2], [11], [12]. The efforts of these and other research teams to increase the mechanical precision and motion control aspects of robotic coordination provide the springboard to higher-level problem consideration by these robotic systems, such as task deconfliction, assignment, and area search. This thesis builds on these previous works by exploring multi-robot systems' ability to link complex behaviors together for complex objective completion. While some researchers have implemented emergent behaviors using biologically inspired algorithms that use simple reactive interaction, our research focuses on highly coordinated planning to achieve deliberative solutions to the problems of task assignment and area search [13], [14].

Search problems can be defined as the exploration of a physical space by sensors in order to observe all points contained within that space. Complete search consists of at least one sensor observation per unit of search area, and an optimal search consists of exactly one sensor observation per unit of search area. This definition translates directly to robotic coverage problems, as described in [15], and much work is being done to advance autonomous systems' ability to achieve solutions to such search and coverage problems. In 2011 researchers from NPS, the University of Southern California, and the University of Minnesota presented autonomous search techniques with specific application to mobile robotics [16]. The search techniques explored in their research involved adversarial game-based utility maximization and probabilistic path cost minimization involving perfect and imperfect sensors. In 2013 researchers from NPS used "mission performance" to evaluate area search patterns used by agents operating within contested areas [17]. This approach to the search problem differs from other work in this area by focusing on conducting the search with counter-agent evasion as a consideration rather than only considering basic search performance measurements that optimize the search coverage [17]. Another effort that looked at metrics other than basic performance measurements to quantify success in search was conducted in 2007 from the California Institute of Technology in [18]. In it, the researchers explored the problem of search for a particular target in the context of the decisions the searcher-agent makes during the pursuit of the target, not just the perceptions received from its sensors [18].

Many of these mentioned works generally explored single-agent searcher configurations that sought to optimally segment the search space and path choices under certain conditions. Works such as [19] focused on multi-UAV coordination dependent on human-handler intervention, making design decisions based on human factors. This thesis, however, explores and enhances multi-agent searching and objective execution by focusing on the agents' ability to communicate and decide amongst themselves how best to segment the search space and deconflict individual path options. To this end, the following research efforts are germane to the area of robot-to-robot autonomous coordination.

Acknowledging the challenges of coordination over lossy communications networks, [20] introduced a decentralized task assignment scheme that assigned multiple agents to multiple moving targets where the agents decided to communicate based on how much of their local information had changed since their last communication. Researchers in [21] approached this problem by using a subset of aerial swarm participants in a "beacon" capacity, loitering and providing information to "explorers," the remainder of the swarm, in order to search indoor corridors and spaces. As explorers moved from beacon to beacon and arrived to an unexplored area, one of the explorers dynamically changed their role to beacon to continue the search [21]. In 2015, NPS used a centralized relationship from one UAV to all other swarm participants to communicate search commands, using what [4] terms a "Teamleader Agent" dynamic, to successfully segment and deconflict search paths within an area search [22].

This thesis seeks to distribute as much autonomy across the multi-robot system as possible during complex behaviors such as area search. Work conducted in [23] by researchers at NPS explored the mission assignment problem among multiple searcher-agents conducting an area search where targets are observed in the environment and handed off to subsequent searchers [23]. Researchers in [7] used agents to conduct search and attack functions on objectives they encountered in a given search space, while agents in [24] add classification and verification to these functions, and [25] adds Battle Damage Assessment (BDA) and the decision to ignore a target to the list. This thesis explores behavior along the same lines as the functions defined in [7], [13], [23]–[25] above. These works present a relative line of demarcation and lineage for this thesis as we explore the limit to which we can decentralize the assignment process and increase the agent-to-agent coordination capability in multi-robot systems with auction algorithms.

Auction algorithms are used for assigning resources to agents in a decentralized manner. They solve assignment problems by presenting opportunities for bidding on elements of a resource pool at certain intervals with certain costs assigned to each element as a function of the desired outcome. In 1979 the definition of an auction algorithm was offered in [26], [27], and a distributed method was introduced for assigning objects to the highest bidder. In [27], the auction process is described as having a bidding phase and an assignment phase. In the bidding phase, all bids for resources are collected by a central auctioneer. In the assignment phase, pairs of bidders and objects are created where no bidder owns more than one resource and no resource is owned by more than one bidder. Decades of work related to solving task assignment problems with auction algorithm implementations now exists.

In [28], a consensus-based auction algorithm is introduced in order to divide and communicate task assignments among agents within a multi-robot system. An interesting result of [28]'s consensus-based auction approach is the removal of the requirement to have an agent act as the auctioneer, removing reliance on a potential single point of failure, assuming inadequate redundancy exists. Research conducted in 2012 at NPS explored the use of an auction algorithm for swarm-on-swarm assignment of targets [29]. In that work, the friendly swarm's utility metric, or cost function, sought to minimize total distance traveled from the current friendly agent's location to the oncoming hostile agent, influencing which friendly agents bid for which hostile agents at any given time in the scenarios [29]. [30] used a tailored combinatorial auction algorithm and a modified winner determination algorithm to conduct multi-agent negotiation for whether or not to participate in a collaborative plan. The authors of [30] used roles (i.e., jobs) within the joint plan as the resources for purchase by the bidders and included each bidder's personal schedule of other activities as private attributes of each bidder object. Their bids considered time for role completion as a constraint to ensure agents did not overtask themselves if they ended up winning their role. In this thesis, we contribute to this body of research by exploring ways in which these principles, and other aspects of auction algorithms, can be applied to autonomous area search problems.

1.4 Thesis Organization

The scope of this research effort includes the application of auction-based algorithms and their utility functions to assignment of cells in an area search. This thesis is divided into five chapters. Chapter 1 provides the motivation for this research, a summary of related efforts, and an overview of the area search problem. Chapter 2 discusses current implementations of auction algorithms and describes how they can be applied to cell assignment during an area search. Chapter 3 provides an overview of our autonomous area search implementation, `AuctionSearch`, and the major branches of execution which create the assignments and conduct the search. Chapter 4 presents the results of our utility functions across search areas and system sizes and analyzes their impact on the efficiency of the area search in simulation and in live-flight testing. Chapter 5 presents our conclusions, findings, and future work that may further illuminate the research area.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Approach

The objective of this thesis is to assess the effectiveness of using auction algorithms with various utility functions in multi-robot systems to assign search cells to individual robots and to autonomously and dynamically complete an area search.

In this chapter we begin to detail how multi-robot systems can link complex tasks together to develop and execute complex courses of action without human intervention. We first lay the groundwork for our work in robot-to-robot coordination with a discussion of the different variations of auction algorithms and their relationship to the generalized assignment problem. We then expand the basic auction algorithm definition for use in a fault-tolerant approach to autonomous area search.

2.1 Methodology

This thesis investigates the research objectives outlined in Section 1.2 in two steps. First, we present an overview of auction algorithms, their purpose, their variations, and discuss their feasibility as a solution for autonomous decision making during an area search by a multi-robot system. Second, we present our auction algorithm implementation, `AuctionSearch`, and the scenario-based experimentation with various utility functions applied. Ultimately we seek to achieve complete, efficient, and fault-tolerant search execution without human intervention.

2.2 Auction Algorithm Overview

The overarching goal of auction algorithms is to assign agents to tasks. The following subsections describe the assignment problem that auction algorithms seek to solve, the advantages and disadvantages of auction-based solutions, and the applicability of auction algorithms to area search using multi-robot systems.

2.2.1 Basic and Generalized Assignment Problems

Fundamentally, the assignment problem seeks to create a one-to-one mapping from a set B of m agents to a set T of n tasks. In the basic assignment problem $m = n$, creating *symmetric assignment* [26]. In the generalized assignment problem, the number of agents does not need to equal the number of tasks, creating *asymmetric assignment* [26]. The goal is to find an optimal distribution of the available agents across the range of tasks [3], [28]. Agents are assigned tasks based on a net-profit function that accounts for the benefit to agent $a_i \in B$ for completing task $t_j \in T$ as well as the cost agent a_i incurs to accomplish task t_j. Solutions to task assignment problems seek to assign every task in T to exactly one agent in B while maximizing the system-wide profit p produced by each agent's net-profit function [27]. For basic assignment, each mapping of agent to task x_{ij} in $B \rightarrow T$ must satisfy the conditions specified by the following linear programming equation [26], [27]:

$$\begin{aligned} \mathbf{max} \sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij} x_{ij} \\ \mathbf{s.t.} \sum_{i=1}^{m} x_{ij} = 1 \\ \sum_{j=1}^{n} x_{ij} = 1 \\ x_{ij} \in \{1, 0\}. \end{aligned} \tag{2.1}$$

For generalized assignment each mapping of agent to task x_{ij} in $B \rightarrow T$ must satisfy the following conditions [26], [27]:

$$\begin{aligned} \mathbf{max} \sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij} x_{ij} \\ \mathbf{s.t.} \sum_{i=1}^{m} x_{ij} = 1 \\ \sum_{j=1}^{n} x_{ij} \geq 0 \\ x_{ij} \in \{1, 0\}. \end{aligned} \tag{2.2}$$

Optimal solutions to the assignment problem can be obtained by centralized or decentralized means, as described in [29]. The term "optimal" is necessarily application specific, as [14] argues that optimal assignment solutions conduct trade-offs between resources, time, and bandwidth requirements [14], [31]. While centralized assignment methods generally require less agent communication than decentralized methods, they frequently lack enough redundancy and dynamism to overcome system failures or changes in operational circumstances [5], [32], [33]. Decentralized methods such as those employing auction algorithms require higher rates of communication among agents but gain the ability to dynamically reallocate assignments as conditions change, increasing the robustness of the system [3], [13].

2.2.2 Auction-based Algorithms

In this section we describe the different auction algorithm variations and the auctioneer mechanisms associated with them. Auction algorithms are a decentralized approach to solving the assignment problem. The goal is to create agent-resource pairs from a set B of m agents and a set S of n resources in a series of rounds. The generic form of an auction creates symmetric assignment, meaning that the number of agents must equal the number of tasks [26]. Each round typically has a bid phase and an assignment phase. The bid phase provides each agent an opportunity to place a bid b for a resource. Each agent maintains a private value v for each resource r in S, and each resource has an associated cost of ownership c. Each agent possesses an amount of money d to spend on purchases of resources. Agents bid on resources that maximize their net value while minimizing their cost incurred. Once all bids are received, the assignment phase completes the assignment of agents to resources for which a winning bid was submitted [27], completing the round.

In the generic form of an auction, each mapping of agent to task x_{ij} in $B \rightarrow S$ is specified by the following linear programming equation [26], [27], [33]:

$$\begin{aligned} &\mathbf{min} \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij} \\ &\mathbf{s.t.} \sum_{i=1}^{m} x_{ij} = 1 \\ &\sum_{i=1}^{m} x_{ij} = 1 \\ &x_{ij} \in \{1, 0\}. \end{aligned} \tag{2.3}$$

Resources assigned in one round can be reassigned in subsequent rounds based on the competing agents' bid values. The auction continues in this fashion until some termination criteria has been achieved. Typical termination criteria for an auction include $|S| = 0$, indicating no remaining resources requiring assignment or $|B| = 0$, where there are no bidders remaining who require resources. Other termination criteria can include no-bid rounds where no bidder accepts the current price of any of the given resources or no bids being submitted within a given time period [34], [35].

In order to achieve optimal assignment of agents to tasks, utility metrics must be used to influence which agents desire to own which resources, maximizing their individual utility while advancing the broader goal. We define "utility" in agents in the same manner as utility based agents in artificial intelligence: agents seek to maximize a hardwired cost-benefit function in order to drive their individual decisions [4], [5], [30], [34], [36]. An auction's parameters and utility functions can be made arbitrarily complex; run-time, degree of understanding of the current situation, and communication bandwidth must all be taken into consideration when determining bidder utility functions. Common utility functions include maximizing profit, minimizing cost, or minimizing aggregate time to complete a set of tasks [32], [35]–[38].

2.2.3 Elements Common to Most Auctions

Many implementations of auction algorithms exist with a wide range of applications, including dividing cloud computing resources as described in [39], completing government procurement [40], consumer credit [34], and exploration of Mars [5], [33] to name only a few. While there are many tailorable attributes and variations of auction algorithms, their implementations have standard components that are generally common to them all. The major structures are listed below:

$$\begin{aligned} R &= \{resource_1, resource_2, ..., resource_j\} \\ resource_j &= (resourceID_j, cost_j) \end{aligned} \tag{2.4}$$

$$\begin{aligned} B &= \{bidder_1, bidder_2, ..., bidder_i\} \\ bidder_i &= (bidderID_i, money_i) \end{aligned} \tag{2.5}$$

$$\begin{aligned} bid_{b,resource_j} &= (bidderID_i, resourceID_j, price_j) \\ price_j &= utility_b(cost_j). \end{aligned} \tag{2.6}$$

There are multiple methods for assigning costs and driving bidder decisions in auctions. In [1], the chosen cost function seeks to minimize the collective time for a set of agents to complete a set of tasks, which they call the "total mission time," weighing the solutions that take the least amount of time to execute the highest. The authors of [13] suggest power consumption as another cost to consider when assigning agents to tasks. In [27] the author describes the primal assignment problem, wherein a bidder holds a particular value for a resource that the bidder wants to maximize with the purchase of it as a byproduct of a bidder-specific utility function.

2.2.4 Single-Item Auctions

The first type of auction is a single-item auction, sometimes referred to as a progressive auction, where bids are placed for one item at a time [5], [35]. Single-item auctions are often "open-cry," meaning the entire set of resources, their costs, and the set of current bids are known to all bidders throughout the entire auction, however this is not a specific requirement [34], [35]. A single-item auction process can be used to solve both the basic and general assignment problems of Equations 2.1 and 2.2 respectively. The goal is to create agent-resource pairs from a set B of m agents and a set S of n resources that satisfies the following conditions:

$$\mathbf{min} \sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$$
$$\mathbf{s.t.} x_{ij} \in \{1, 0\}. \tag{2.7}$$

The basic single-item auction-based assignment algorithm as described in [8], [26], [27], [29], [37] is provided in Algorithm 1:

Algorithm 1 Algorithm for Conducting a Single-Item Auction

```
S ← [(resourceID_0, cost_0), ... , (resourceID_m, cost_m)]
B ← [(bidderID_0, money_0), ... , (bidderID_n, money_n)]
for j = 0 to length(S) do
    bids ← []
    highBid ← 0
    myBid ← 0
    winner ← NULL
    for i = 0 to length(B) do
        utility_ij = bidder[i].utility(resource[j])
        if utility_ij > myBid then
            myBid = bidder[i].calc_bid(resource[j])
        end if
    end for
    if myBid > 0 then
        bids.append(bid_ij)
    end if
    for k = 0 to length(bids) do
        if auctioneer.winner_determination(bid_k) > high_bid then
            high_bid ← bid_k
            winner ← bidder[k]
        end if
    end for
end for.
```

To begin the auction, the auctioneer needs two critical pieces of information: $|S|$, the number of resources to be auctioned, and the cost c_j for each resource $r_j \in S$. At a minimum, the auctioneer must communicate c_j for each $r \in S$ to all bidders $b \in B$ prior to bidding, unless a specific application benefits from tailoring this to include blind bidding, where c_j is unknown.

In order for bidder b_i to gain possession of resource r_j, b_i must first submit an allowable bid q and subsequently be chosen as the winner of the round by the auctioneer. An allowable bid is any q that conforms to all prescribed constraints of the auction. For example, $q = allowable\ \mathit{iff}\ q \geq c_j + \delta$ is an auction rule that indicates that all bids must exceed the current price for r_j by at least δ. [35] and [41] call this a "minimum increment" rule. A bid of $q = 0$ is considered a *no bid*, and can either be viewed as a trivial case or as a means for a bidder to explicitly abstain from bidding for a given resource [42]. Further, a useful (and necessary) constraint is $q \leq m_{b_i}$, forcing allowable bids to be ones which bidders can actually afford resources for which they are selected as winners. Many other application specific constraints are possible as well [5], [34].

Once each bidder has had the opportunity to bid on a given resource r_j, the auctioneer awards it to the winning bidder b_i based on the auction's specification, ending the round. In the simplest version of winner determination for single-item auctions, a bidder b_i wins a resource r_j if b_i submitted an allowable bid q_{b_i} to the set of bids Q such that $r_j \rightarrow b_i\ \mathit{iff}$ $q_{b_i} = max(Q)$. Once a winner has been selected, the single-item auction continues in this manner until some termination criteria is triggered.

2.2.5 Combinatorial Auctions

Single-item auctions have the advantage of fine grained control over resource distribution, however if the number of resources is substantial it may take unacceptably long to create complete assignment of agents to resources [27]. The time-complexity of assignment increases polynomially with the number of possible agent-resource pairs [1]. Combinatorial auctions seek to achieve complete assignment more quickly by assigning variably-sized subsets of the overall set of resources to each agent. The goal is to assign each agent in a m-sized set B to a k-sized subset of resources T, such that $c_i\colon T \rightarrow \mathbb{R}$. C is a set of this mapping of agents to task-subsets x_{ij} in $B \rightarrow T\sim$ is subject to the linear programming equation [43]:

$$\begin{aligned} \textbf{max} \quad & \sum_{i=1}^{m} c_i T_i \\ \textbf{s.t.} \quad & T_i \bigcap T_j = \emptyset \quad \forall i \neq j \\ & \bigcup_{i=1}^{m} T_i = T. \end{aligned} \tag{2.8}$$

A combinatorial auction follows the same general structure as a single-item auction except that bidders are allowed to pursue any subset of resources $T \subseteq R$, known as bundles or packages, with a single bid [32]–[34], [43]. In turn, each bidder $b \in B$ places a set of bids U for T with the following forms:

$$T = \{resource_1, resource_2, ..., resource_j\} \tag{2.9}$$

$$U_{bT} = \{bid_{resource_1}, bid_{resource_2}, ..., bid_{resource_j}\} \tag{2.10}$$

$$\begin{gathered} bid_{(b,resource_j)} = (resourceID_j, price_j) \\ price_j = utility_b(resource_j). \end{gathered} \tag{2.11}$$

The form and application of combinatorial auctions is discussed in [8], [30], [32], [33], [37], with a version of the auction's general form given by Algorithm 2:

Algorithm 2 Algorithm for Conducting a Combinatorial Auction

```
S ← [(resourceID_0, cost_0), ... , (resourceID_m, cost_m)]
B ← [(bidderID_0, money_0), ... , (bidderID_n, money_n)]
bids ← []
U = []
highBid ← 0
winner ← NULL
for i = 0 to length(B) do
    U_iT = bidder[i].utility(S)
    if length(U_iT) > 0 then
        bids.append(U_iT)
    end if
end for
for k = 0 to length(bids) do
    if auctioneer.winner_determination(bid_k) > high_bid then
        high_bid ← bid_k
        winner ← bidder[k]
    end if
end for.
```

For combinatorial auctions, finding the optimal assignment of subsets of resources to bidders that maximizes utility, or the winner determination problem, is known to be NP-complete and must therefore be combinatorically constrained to achieve tractability [5], [13], [32]–[35], [44]. In combinatorial auctions, the most basic version of winner determination is accomplished by the auctioneer selecting the bid set U_b that either maximizes utility or sells most of the available resources:

$$winner = max(\sum_{i=1}^{m}\sum_{j=1}^{n} q_{b_i} + c_j) \cup max(\sum_{i=1}^{m}\sum_{j=1}^{n} |U|) \quad \text{where } q_{b_i} \in U_b. \tag{2.12}$$

Combinatorial auctions can create efficient assignment solutions as the size of the bid sets grow, creating shorter auctions overall as more resources are consumed. They can also become cumbersome, however, if bidders' utility functions are overly complex (e.g., more than simply maximizing some value associated with each resource). The complexity of the Combinatorial Auction Problem (CAP), an instance of the well studied Set Packing Problem (SPP), is an NP-hard problem which deals with the complexity introduced when a bidder must consider all possible subsets of resources to find an optimal combination [43]. The Multi-Dimension Multiple-Choice Knapsack Problem (MMKP) is a similar problem wherein each bidder must choose single elements from multiple resource pools.

A common thread through these problems is the combinatorial explosion that occurs as the size of the resource pool grows [5], [32]–[35]. While complex utility functions effect single-item auctions as well, the deliberation and analysis an agent might do while selecting a set of resources can increase exponentially compared to a single resource [33]. If bidders are spending inordinate amounts of time deciding what combination of resources best maximizes their utility, the auction may fail to achieve complete assignment in a timely enough manner. As computing power has increased over the years, so has the ability to implement more complicated versions of combinatorial auctions, however the optimality problem is yet to be solved [35], [43].

To combat the complexity of formulating optimal bid sets on the bidder's side and selecting the optimal winner on the auctioneer's side, winner determination algorithms must be carefully tailored to the particular application in order to create efficient solutions [30], [42]. Some useful heuristics include limiting the size and number of bundles allowed in the auction and using efficient clustering algorithms to produce bid sets [5], [32].

2.2.6 Centralized Auctioneer Mechanisms

Determining which agent has submitted the winning bid among the set of bids for a particular resource is the critical function linking the bid and assignment phases of an auction round. Winner determination is either conducted by a central auctioneer or by a decentralized linear program executing exchanges of resources between individual agents [26], [27].

In centralized implementations, the role of auctioneer is either statically assigned to one of the participating agents or it can be rotated among them. Agents bidding on resources submit their bids to the auctioneer who then selects the winner based on the set of received bids and the auction's specification (e.g., highest or lowest bid). Some implementations include the auctioneer as a participating bidder while others exclude the auctioneer for the duration of the auction [38], [43]. When the auctioneer receives identical bids for a given resource, the winner is typically determined randomly by the auctioneer, unless the auction is designed to avoid such situations [26], [43].

The most obvious limitation associated with using a centralized auctioneer is the reliance on a single point of failure. If the auctioneer experiences a loss of functionality then the auction may fail to properly execute. Creating redundancy would increase resilience but would exacerbate or introduce other problems, such as data consistency and bandwidth demand. Walrasian methods, discussed in [41], [43], attempt to reduce the impact of centralization by replacing the selective auctioneer with a more passive price-setting merchant, but the reliance on a singular entity remains.

2.2.7 Distributed Auctioneer Mechanisms

Decentralized implementations are generally less complex and more resilient to failure than centralized ones. In decentralized implementations, the role of auctioneer is distributed among the agents and each agent is capable of both bidding and auctioneering. To start, each agent iterates through each resource and identifies the one that achieves the highest gross utility given the agent's utility function. Once identified, the agent then bids and greedily assigns itself to the resource, exchanging its current resource for the higher grossing one. If multiple agents are competing for the same resource, the price is increased with each bid placed until there is only one agent remaining whose gross utility is still maximized. The auction continues in this fashion until every agent is "happy," meaning every agent is assigned a resource that maximizes its gross utility [26], [27].

2.3 Application of Auction Algorithms to Autonomous Area Search

In this section we begin to discuss the application of auctions to the complex task of area search. We start by tailoring the terms used in the preceding sections to the search problem. Secondly we incorporate auctions into an area search algorithm and discuss the various cost functions that can be utilized. Then we detail how auctions factor into an area search algorithm including the fault-tolerance gained with dynamic reallocation.

To start our discussion we define our terms for using auctions in area search applications. We continue to use the term “auction” for describing the action of bidders bidding for resources for simplicity’s sake. We use the following terms and definitions from this point forward:

1. Search Area: The predetermined physical area that the agents are required to explore.
2. Search Space: The search area broken down into an undirected graph of cells by some cellular decomposition method (e.g., trapezoidal, grid, boustrophedon).
3. Cell: A biddable and awardable resource that represents a geometric subset of the search space. Cells are organized as a set of waypoints distributed based on the owning agent’s sensor characteristics.
4. Waypoint: An element of a cell that represents a physical location that an agent must travel to in order to be considered explored. The dispersion pattern of the waypoints should be a result of the dynamics of the configuration space, such as sensor sweep width, speed, and turn radius of the searcher.
5. Searcher: An agent assuming search responsibilities of cells for which it has bid for and won. Searchers participate in auctions and communicate with other searchers.
6. Auctioneer: A centralized or decentralized mechanism for determining which agent won which cell. The position of auctioneer is typically accomplished by a single agent, possibly a searcher [28], [32], [33]. In our implementation described in Section 3 we explore methods that reduce or remove this control and communication bottleneck [41], [43].

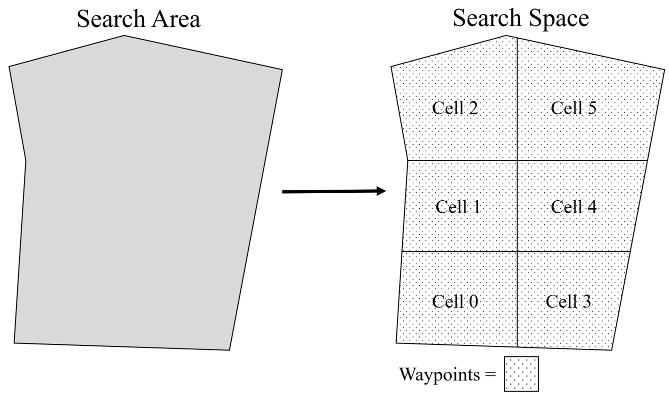

Figure 2.1. Search Area Discretized into Search Space

The application of auction algorithms to the area search problem is fairly straightforward. In a centralized scheme, the auctioneer first acquires the lists of available bidders, cells, and their associated costs. In a decentralized scheme, the agents must first send each other cell and bidder information. Next, the searchers receive and verify the list of available cells (with costs) via transmissions from some decentralized formation control system [45] before utility calculation and bidding.

Agents place bids for cells according to their individual utility functions and are assigned cells for which they submitted the winning bid. Once assigned a cell, searchers move to and systematically explore the cell's waypoints, conducting new auctions for follow-on cells as required, until the search is complete [32], [33]. In order for the search to be considered complete, every waypoint of every cell in the search space must be explored by a searcher. The general form of an area search using auctions is presented in Algorithm 3 and in Figure 2.2 [33]:

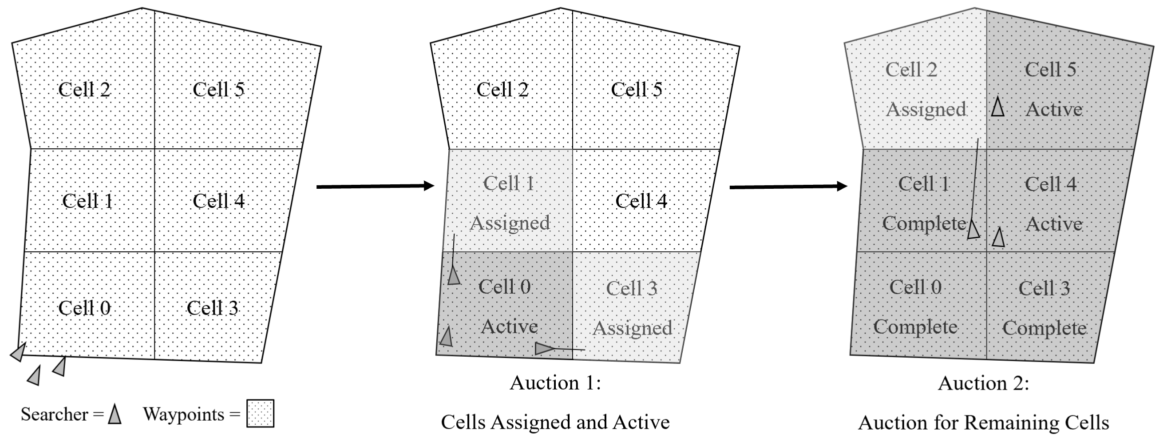

Figure 2.2. Area Search Execution Using Auctions

Algorithm 3 Algorithm for Conducting an Area Search with Auctions

---

$searchers \leftarrow [(searcherID_0, money_0), \ldots, (searcherID_m, money_m)]$

$search\_space \leftarrow [(cellID_0, cellStatus_0, cost_0), \ldots, (cellID_n, cellStatus_n, cost_n)]$

while not $search\_complete$ **do**

  $cell\_assignments \leftarrow conduct\_auction(search\_space, searchers)$

  while not $cells\_complete$ **do**

    $searchers.search(cell\_assignments)$

  end while

end while

2.3.1 Utility Function Considerations

The agent's utility function determines what the agent values in being assigned a given task. Many factors can contribute to the calculation of such value. Considerations of interest include distance, remaining power level (i.e., endurance), agent type and capabilities (e.g., quad-copter or fixed-wing), speed, and agent sensor sweep-width, to name a few. A more detailed discussion of these considerations is offered in Chapter 3.

Other research efforts have also explored these considerations. Prim Allocation, introduced in [33], uses the distance of the cheapest previous bid in each agent's bidding history to influence its utility. [32] used a bidding strategy that included the cost of a bundle of waypoints plus exactly one dollar per every unit of distance the agent was from each waypoint as the bidding strategy. In [46], the searchers' utility is based on their ability to action objectives sooner rather than later, with weights assigned based on the length of time each objective takes to complete. This stratification made it possible for searchers to be assigned tasks for which they had an adequate amount of power remaining to accomplish, preserving a high utilization rate. [25] notes that greedy first-step assignments in a search's first auction are generally unavoidable, given utility functions incorporating distance from a given waypoint or cell.

An important aspect of auctions as applied to area search versus other applications is that the most important goal is complete assignment, or *tatonnement*, of cells and waypoint coverage over monetary frugality [41]. Deeply sub-optimal solutions result from cells going unpurchased for long periods of time, as costs associated with unpurchased cells grow as the search progresses farther away. Further, there is no chance of complete coverage if cells go unpurchased indefinitely. With achieving complete assignment our primary goal, bidding strategies need not necessarily save money, and searchers can be provided new total amounts of money for each auction in order to avoid such situations.

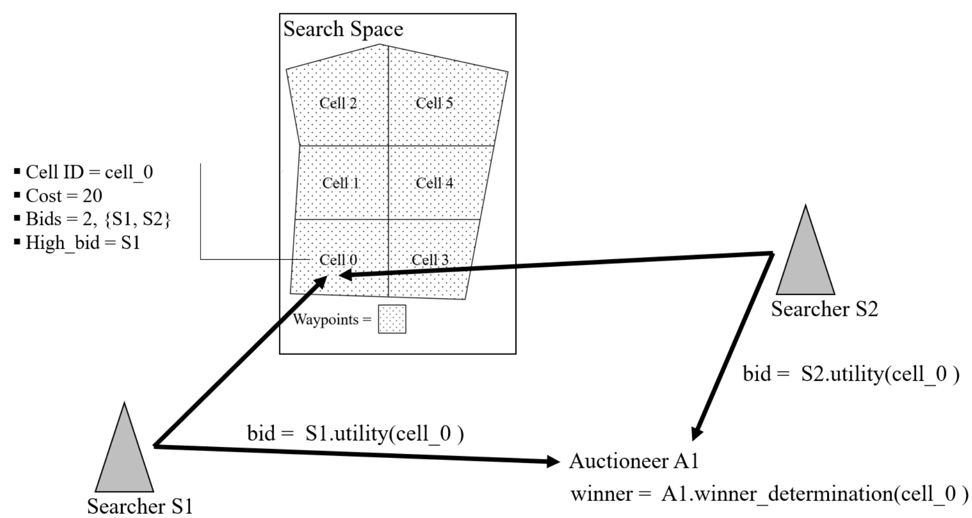

Figure 2.3. **Bidding for Cells.** Agents submit bids in similar ways whether a centralized or decentralized scheme is used. Some auctioneer mechanism determines which agent won the given cell.

Arguments have been made that combinatorial auctions are better suited to producing optimal assignment solutions than their single-item auction counterparts [32], [35]. It is argued that they produce optimal sub-teams during search operations as compared to general auctions because they optimize the use of each agent's "synergy," as [32] describes it, relative to the bundle of resources it bid for and won. The authors of [32] and [5] define this "synergy" as the advantage gained from selecting two or more cells that are close together in a single bundle rather than bidding for one of the cells, winning it, then bidding on the second cell, losing it, resulting in potentially sub-optimal collective search times [32].

Auctions are only useful for assignment during area search if the status of the various data structures is kept current and accessible to the necessary agents. The auctioneer should only offer unique, unexplored cells for bidding or risk missing or duplicative coverage resulting in sub-optimal results at best.

2.3.2 Search Space Maintenance

Before each auction round, the searchers need to know which cells are up for auction and which ones are not. In order to do this efficiently, the searchers need to track the state that each cell is in. The set of possible states that a cell can be in and the transitions to and from those states must be well-defined to ensure accuracy. We define "maintenance" here as deciding how the search area is divided into cells and tracking what the current state of each cell is. When combined, this provides a snapshot of the overall state of the search at a particular point in time. [25] presents a well tuned set of possible states to consider. The states used in [25] are presented below:

$$A = \{available, associated, assigned, active, complete\} \tag{2.13}$$

where *associated* relates to a "provisional" assignment and *assigned* refers to an actual assignment that translates into adjustment to robot motion control [25]. Defining, assigning, and communicating these states, and doing so efficiently, is a chief concern because they represent the direct input and output of any area search algorithm. Further, if dynamic applications are used, as described in the next subsection, then ensuring accurate input to each auction is vital to avoid detrimental error propagation.

Search space maintenance can be distributed or centered on a single searcher. [22] discusses the relative advantages and disadvantages of each method. Managing a central search space requires a ground station or searcher to be assigned as the search space manager. The search space manager is responsible for receiving statuses from searchers, updating the search space, and rebroadcasting the updated information.

In decentralized maintenance, each searcher maintains its own current understanding of the search space over the course of the search. The search area coordinates are first issued to each searcher followed by execution of the same cellular decomposition, adjacency development, and waypoint distribution algorithms across all of the searchers. While this presents a duplication of effort and requires inter-swarm update messaging, it avoids reliance on a single point of failure.

Regardless of which search space management solution is chosen, network communication bandwidth and update frequency must be sufficient to maintain accuracy and universal understanding. Consensus algorithms such as those analyzed in [31], lazy and eager consensus introduced in [47], Kalman Consensus in [48], and consensus-based auction algorithms in [28] have been shown to be viable communication solutions in dynamic and lossy network environments [49].

2.3.3 Dynamic Search Space Reallocation Via Auctions

A key advantage to conducting an autonomous area search with auctions is the potential for dynamic reallocation of search cells [33]. We define "dynamic reallocation" here as updating the searcher-cell assignment solution given current statuses for the searchers and cells. At the outset of a search but after all preliminary cellular development is completed, an auction is started to create initial assignment of cells ranging from one up to and including complete assignment.

Dynamic reallocation occurs when some trigger is met during the search that indicates a new intermediate auction needs to take place given current information. An intermediate auction is any auction occurring after the initial assignment auction has taken place. Figure 2.4 depicts how an area search progresses including dynamic reallocation and intermediate auctions.

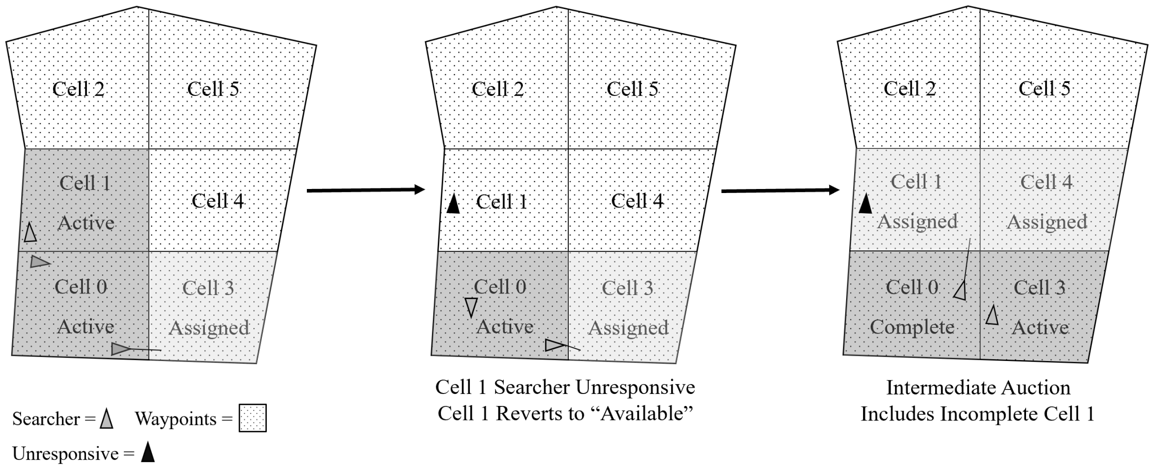

Figure 2.4. **Dynamic Auction Application.** This figure shows how a system of searchers can recover after one of the searchers leaves the search. Once the remaining agents sense the loss, they can reallocate the lost agent's work to operational searchers.

Algorithm 4 Algorithm for Conducting an Area Search with Dynamically Applied Auctions

$searchers \leftarrow [(searcherID_0, money_0), \ldots, (searcherID_m, money_m)]$
$search\_space \leftarrow [(cellID_0, cellStatus_0, cost_0), \ldots, (cellID_n, cellStatus_n, cost_n)]$
for $i = 0$ to $length(search\_space)$ **do**
  if $search\_space[i][cellStatus]$ not in $\{active, complete\}$ **then**
    $cells\_to\_auction.append(search\_space[i])$
  end if
end for
while not $search\_complete$ **do**
  $cell\_assignments \leftarrow conduct\_auction(cells\_to\_auction, searchers)$
  while not $cells\_complete$ **do**
    $searchers.search(cell\_assignments)$
    if $searcher\_reported\_out$ **then**
      $cells\_to\_auction.update(search\_space, searchers)$
      $cell\_assignments \leftarrow conduct\_auction(cells\_to\_auction, searchers)$
    end if
  end while
end while.

Triggers include cell completion, searcher-agent failure, or any activity that causes a searcher to exit the search, such as encountering some higher objective as in Figure 2.4. Regardless of which event causes the trigger, the result is still the net loss of an agent (or agents) responsible for searching a cell. The cell's status is then reverted from *active* to *available* using Equation 2.13 and the auction re-initiates. Algorithm 4 includes the cell statuses from Equation 2.13 and a Boolean test to check for triggers.

Dynamic reallocation is advantageous because variability permeates all aspects of robotics, and the more flexible robotic systems are to changing environmental conditions the better they are at managing real world problems such as search when other objectives compete for priority. Determining which triggers re-initiate assignment (i.e., which triggers make $searcher\_reported\_out$ == True) will affect auction frequency, completion time, and individual robot utilization scores [5].

2.4 Summary

In this chapter we presented an overview of auction algorithms, discussed how they create solutions to the assignment problem, and explored their applicability to autonomous area search. When applied to area search, auction algorithms create agent-cell pairs where the goal is to minimize the total system cost required for completing the search while maximizing overall system utility. Auctions achieve this by using individual-searcher utility functions which seek to greedily minimize individual cost to search cells, managing run-time concerns by using the simplest functions possible that still achieve the best possible assignments. Auction algorithms also allow for dynamic reallocation of cell assignments at certain intervals and given certain triggers which allow the system-wide costs and utilities to be reshuffled with current state information taken into account.

This chapter also discussed many approaches to auction implementations and the different impacts and advantages associated with using single-item and combinatorial auctions to create agent-cell assignments. Single-item auctions, where a single cell is bid for by each agent, typically allow for locally optimal assignments because each agent bids highest for the cell that achieves the agent's highest utility.

Sub-optimal search completion times can occur, however, if agents are only associated with a single cell because they must continuously wait for follow-on cells to be assigned via auction. This inefficiency can be mitigated with various auction-trigger strategies. Combinatorial auctions can achieve faster search completion times since multiple cells are bid for in bundles of high-utility cells at the cost of computation complexity that can likewise hinder completion times.

Both types of auctions can assign cell-agent pairs using auctioneer winner determination in both centralized and decentralized fashions, however decentralized methods are far more robust to system failure while centralized mechanisms offer lower communication bandwidth use. All of these factors are taken into consideration in Chapter 3 where we introduce `AuctionSearch`, our implementation for area search using single-item auctions and a decentralized auctioneer mechanism.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Implementation and Experiment Design

In this chapter we explain `AuctionSearch`, our auction-based area search implementation. First, we introduce the `AuctionSearch` flow of execution and the search environments used for testing. We then describe our implementation's major elements with specific focus on utility calculations, bid generation, winner determination, and cell assignment and reassignment. Lastly, we explain our use of speed and endurance to derive individualized cell utility values to influence bidding strategies. In Chapter 4 we introduce our speed and endurance utility functions and analyze our implementation's performance in achieving auction-based assignment in an efficient manner.

3.1 `AuctionSearch` Top-Level Flow of Control

In this section we explain the high-level flow of control for `AuctionSearch`. Each agent participating in the `AuctionSearch` behavior executes the algorithms depicted in the flow diagram of Figure 3.1. The two major branches of execution are `IS_SEARCH_AUCTION` and `IS_SEARCHER`. These Boolean-controlled gates are tested each update cycle and executed accordingly. Boolean controlled gates are more suitable than state-based control in this implementation because they support parallel execution of both branches. That is, an agent can execute the search of a cell while also participating in an auction for future cells.

The `IS_SEARCH_AUCTION` branch controls all activities related to cell assignment. In this branch agents update their understanding of search progress, generate utility values for each cell, calculate and submit bids for their favorite (i.e., highest utility-gaining) cells, and conduct auction round winner determination. This branch ceases execution when each agent has their required number of cells assigned or there are no more cells left to auction. The branch flow is depicted in Figure 3.1.

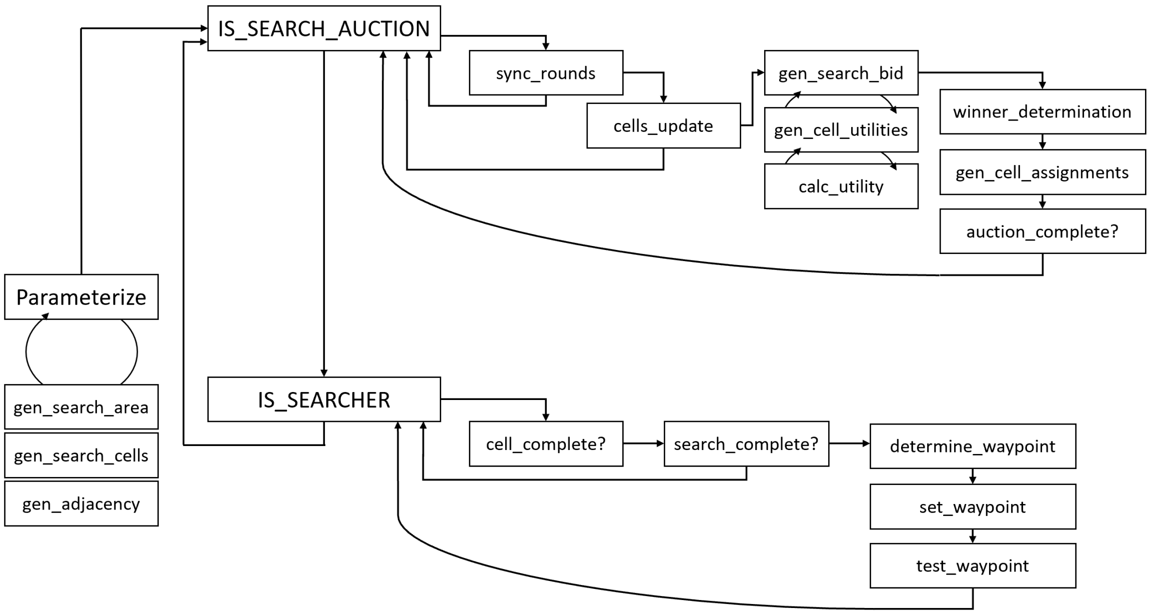

Figure 3.1. **AuctionSearch Flow of Control.** The overall flow of AuctionSearch is controlled by the two major Boolean-controlled gates, IS_SEARCH_AUCTION and IS_SEARCHER. The algorithm terminates when there are no more cells to search.

The IS_SEARCHER branch controls all search-related activities. Agents who have bid for and won cells execute the search of their assigned cells by following a self-generated series of waypoints. If an agent finishes searching its current cell, it initiates a new auction with all other searchers to complete a new round of cell assignments. If any agent has already received an auction-start message with a current auction identifier, it does not send new auction start messages in order to avoid race conditions.

This branch is no longer executed when the agent has no assigned cell. If there are no cells left requiring search, the AuctionSearch algorithm terminates. Algorithm 5 describes the overall flow of control for our implementation. Each agent executes Algorithm 5 independently.

Algorithm 5 Top Level Control Algorithm for `AuctionSearch`

```
while cells left to search > 0 do
    if IS_SEARCH_AUCTION then
        Execute auction for cell assignments
    end if
    if IS_SEARCHER then
        Execute movement to and search of assigned cells
    end if
end while.
```

3.2 Search Area Decomposition

Before the agents are capable of executing either one of the branches they must first have a common understanding of the area to be searched and the cellular breakdown. As the behavior is initialized, each agent independently breaks the search area down into cells and graphs their adjacency. This process is conducted in a deterministic manner so that all agents generate the same set of search cells and adjacency graphs.

At a minimum, a finite geographical area consisting of at least one cell (which covers the entire area) is required in order to have an agent or group of agents complete an area search. We developed three environments for testing our `AuctionSearch` implementation: a basic search area, a large-basic search area, and a complex search area. The major differences between the two basic search areas and the complex area is cell uniformity and the presence of obstacles.

3.2.1 Basic Search Area

The basic environment is a small rectangular search area containing no obstacles that is broken down into 12 uniform quadrilateral cells. The basic search area pictured in Figure 3.2 was used for algorithmic development in the Software-in-the-Loop (SITL) simulation environment and for live-fly field testing of our design. It afforded the maximum number of iterations and ensured containment within a mandated geo-fenced region of the test site. Working with the basic search area allowed for small-scale tuning of the algorithms in the minimum amount of time and enabled live-fly capability within testing constraints.

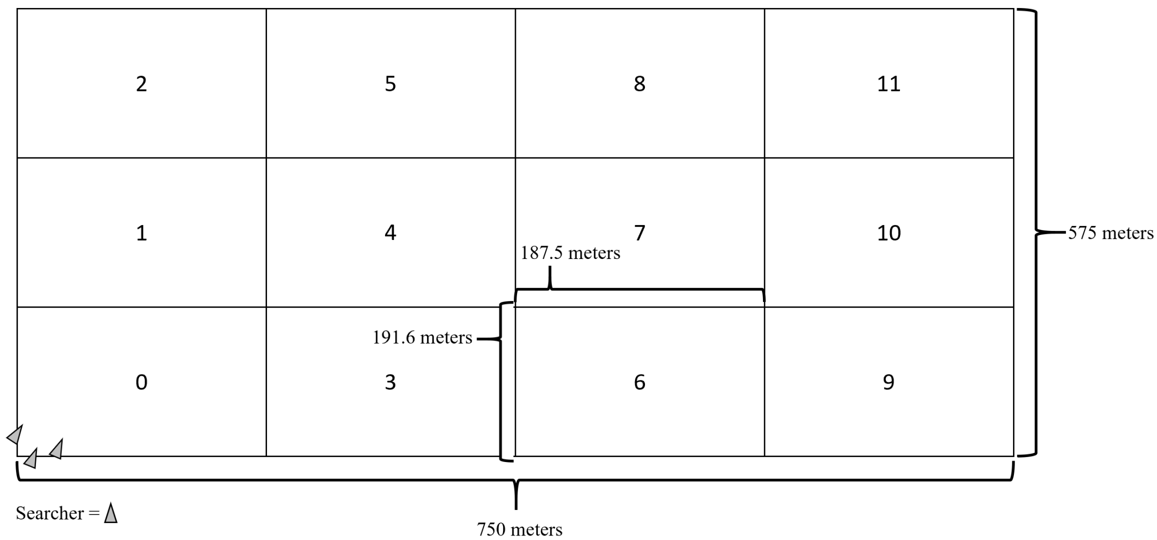

Figure 3.2. Basic Search Area after Grid-Cellularization

The even distribution of cells in the basic search area provides the ability to test cases in which high adjacency exists, such as in cell 4's case in Figure 3.2 or cases wherein multiple agents complete their in-progress cell at nearly the same time (assuming the same start time). Further, the basic area's cell uniformity minimizes the number of cases in which utility calculations are based primarily on cell size and maximizes dependence on speed and distance to the target cell. While cell size and distance to the target cell both ultimately contribute to an overall distance calculation, the basic search area allows us to isolate each variable and observe the contribution of specific independent variables to agent bid values and the resulting assignments.

3.2.2 Large-Basic Search Area

The large-basic environment, pictured in Figure 3.3, is a scaled-up version of the basic search area. It consists of a large rectangular search area that is broken down into 80 uniform quadrilateral cells and it also contains no obstacles. Using a larger area affords the opportunity to observe large numbers of agents in execution and to observe how algorithm run times and cell assignments scale with both the size of the swarm and the number of cells. The shaded region in the lower-left of Figure 3.3 shows the basic search area from Figure 3.2 to illustrate the scale difference between the two.

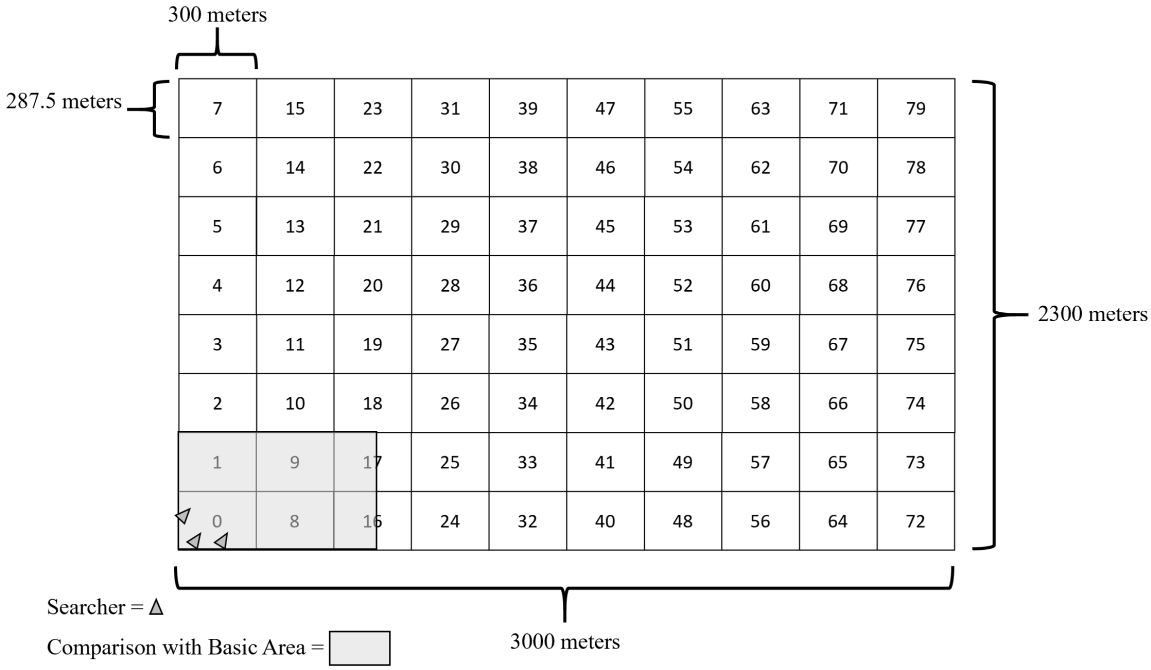

Figure 3.3. Large-Basic Search Area after Grid-Cellularization

3.2.3 Complex Search Area

Many realistic search areas can be represented by a grid of uniform cells canvased across an open area, such as search and rescue or reconnaissance in unobstructed areas. Other realistic search areas may include obstacles or restricted areas where we are not interested in having search conducted, making the environment more complex. In both cases, auction algorithms are a suitable means for assigning searchers to cells. In order to observe how assignment solutions differ from basic to complex environments, we created the complex search area depicted in Figure 3.4.

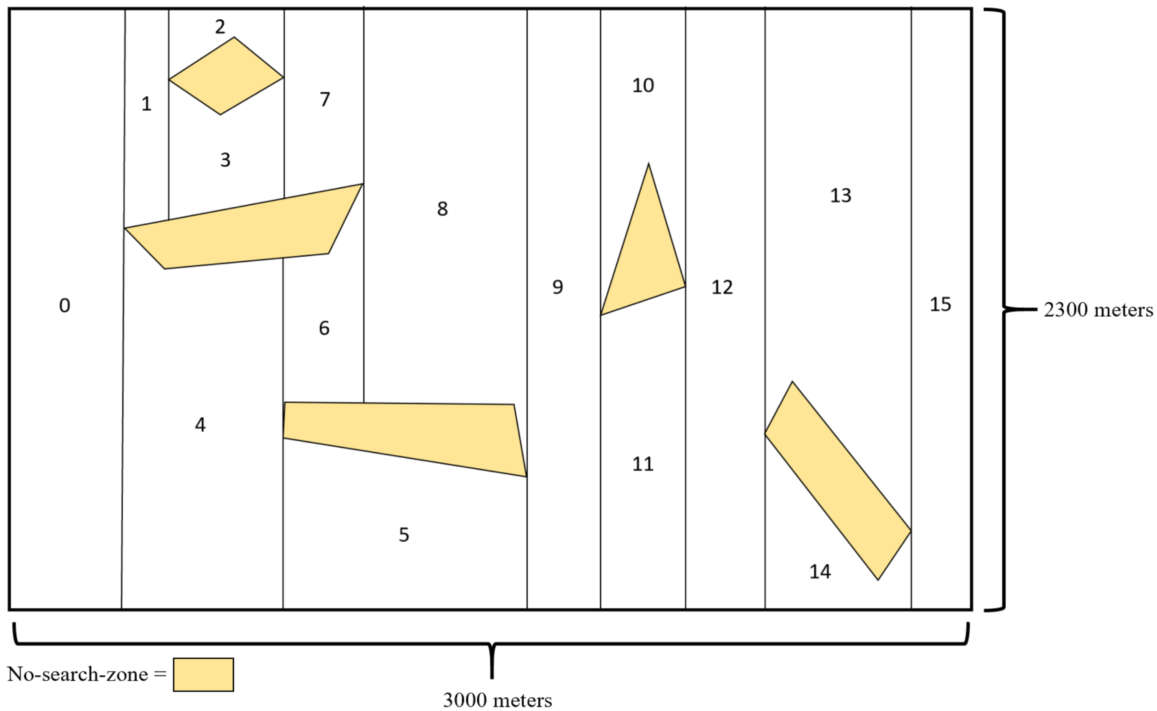

Figure 3.4. Complex Search Area after Boustrophedon Cellular Decomposition

The major difference between using auction algorithms for cell assignment in basic search areas versus complex search areas is that obstacles create non-uniform cell sizes which then impact cell utility calculations. Our implementation uses a Boustrophedon cellular decomposition algorithm as described in [50] that breaks the complex environment down into cells based on left and right critical vertices of the obstacles in the space. The various cell sizes affect the utility calculation for a given cell because a large cell contains more waypoints and takes longer to search than a smaller one.

Another difference between searching a complex and a basic area is the tendency for bottlenecks in the adjacency of the cells. Bottlenecks can occur in both basic and complex search areas, but they are more prevalent in complex areas where obstacles and restricted areas can channel movement between cells. Bottlenecks in less complex areas, on the other hand, are usually a byproduct of agent decision making.

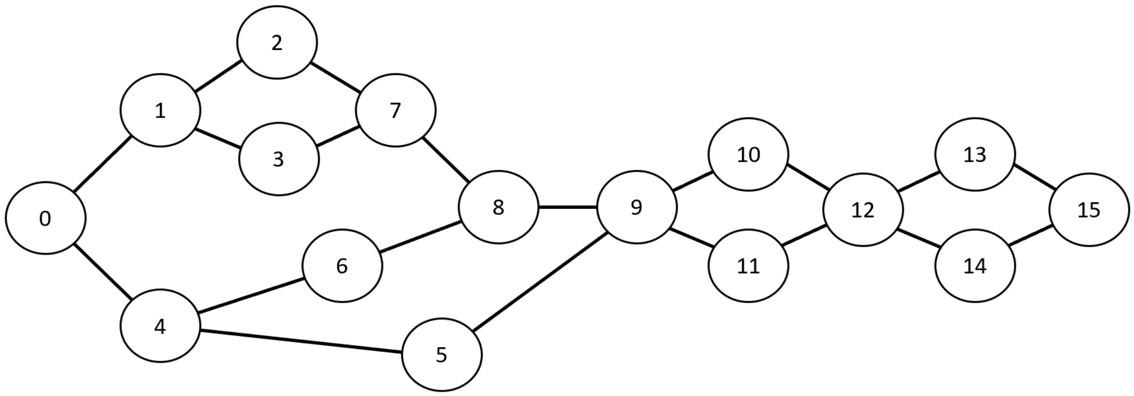

Figure 3.5. **Complex Search Area Adjacency Graph.** The adjacency graph for our chosen complex environment. Adjacency graphs for complex search areas can contain many bottlenecks that effect overall system utilization.

3.2.4 Search Cell State Labeling

Regardless of whether the search area is basic or complex, the set of possible states that each cell can be associated with at any given time is the same. Similar to how [25] categorized cells as "available, associated, assigned, active, and complete," in Equation 2.13, we establish the set of states for our implementation as follows in Equation 3.1:

$$states = \{available, assigned, in\_progress, assignment\_removed, complete\} \quad (3.1)$$

The list below describes each of the possible states given in Equation 3.1 and Figure 3.6 depicts the same states. Cells can never be in more than one state at any one time. Further, they are tracked in our implementation as an enumeration in ascending order so agents can easily detect and log cell state changes reported by other agents by simply identifying a state which is associated with a higher enumeration than what they are tracking.

1. *available*: A cell which is unexplored and unclaimed by any searcher. Available cells are always included as biddable and winnable resources in auctions.
2. *assigned*: A cell which has a searcher associated but has not yet begun to be explored. Assigned cells are included as resources in auctions. An agent who submits a higher bid for an already-assigned cell will assume the assignment, and the losing agent will

relinquish their assignment.

3. *in_progress*: A cell which has a searcher associated and has begun to be explored by that searcher. In-progress cells are never included as resources in auctions. In the event that an in-progress cell's searcher leaves the search, due to malfunction or some other reason, the cell's status is changed to *assignment_removed*.
4. *assignment_removed*: A cell which has been assigned any one of the above mentioned states previously but has since become unassigned. Prior to the next auction, all cells with a status of *assignment_removed* are transmitted to every other agent so that all locally maintained cell dictionaries can be updated to *available*. This intermediate step is helpful because it places cells which are being "thrown back" into the auction into an easily detectable state rather than immediately changing the cell back to *available*. The *assignment_removed* state occupies a higher enumeration than the *available* state, so each agent can detect these cells during cell status updates without having to examine every cell to see whether it is still available or not.
5. *complete*: A cell which has had all of its associated waypoints visited by a searcher. Complete cells are never included as resources in auctions.

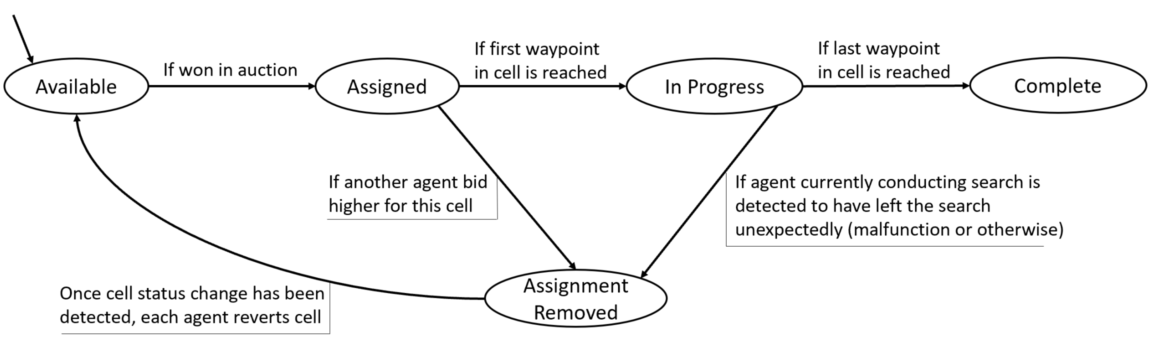

Figure 3.6. **Cell State Diagram.** This figure shows the cell state transitions associated with our auction-based area search implementation. Cells can only be in one state at a given time. Our implementation enumerates these states in ascending order based on Equation 3.1 to enable easy identification of cell status changes.

In our implementation, cells are maintained as objects with certain characteristics. Each agent maintains a dictionary of the cells and their current understanding of each cell's state. Before and after each auction, each agent communicates which cells they observed change in and what those changes were to allow other agents to update their understanding.

During these inter-robot cell status updates, each cell is represented and communicated as a four-tuple in the following form:

$$cell\_status = [cell\_id, cell\_state, cell\_owner, cell\_cost]. \tag{3.2}$$

The *cell_status* field in Equation 3.2 relates to the cell status states listed in Equation 3.1, which are listed in order of ascending precedence with regard to inter-robot updates. Put differently, an update from a given agent that indicates a given cell is *complete* will supersede an update from a different agent that indicates the same cell is only *in_progress*. Continuing this example, an agent receiving this update would change their local understanding of this particular cell's status to *complete* and adjust its data structures accordingly to account for the newly identified completed cell. This process occurs before and after each auction, and auctions are not permitted to proceed unless all participating agents have updated their understanding of cell statuses. As a final note on cell statuses, when all agents are in agreement that the entire set of cells to search are *complete*, the search is terminated.

In the following sections we describe the two major branches of `AuctionSearch` execution that were introduced in Figure 3.1, their algorithms, and the design trade-offs that shaped the implementation.

3.3 Assignment of Search Cells via Auction

In this section we describe the first major branch of execution in our `AuctionSearch` implementation, the `IS_SEARCH_AUCTION` branch. `IS_SEARCH_AUCTION` uses single item auctions to create agent-cell assignment pairs. Figure 3.7 shows a detailed and zoomed in flow of execution for the `IS_SEARCH_AUCTION` execution branch.

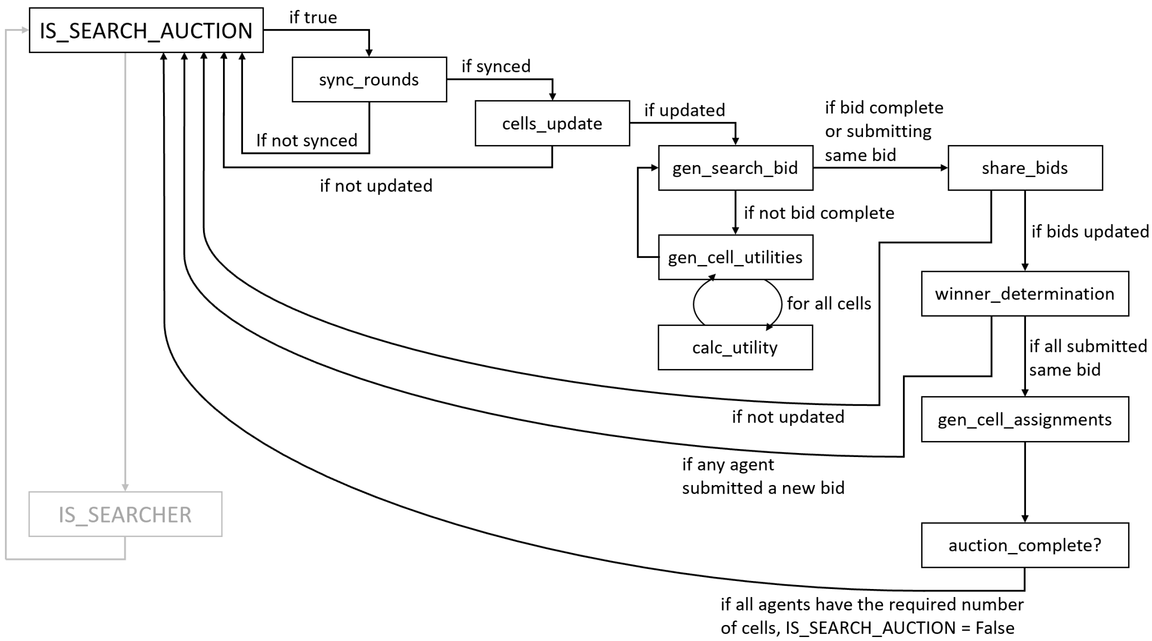

Figure 3.7. **Auction Control in `AuctionSearch`**. `IS_SEARCH_AUCTION` Boolean-controlled execution branch flow of control.

The overall objective of this branch is to generate one-to-one mappings of searchers to cells based on each agent's calculated utility for each cell. A number of `AuctionSearch` class methods support this objective, doing everything from ensuring all agents are operating on consistent data to round winner determination. At the top level, we ensure that only one auction is occurring at any given time by having all agents check whether they are already participating in an auction before they initiate one. If an agent triggers our new-auction criteria (e.g., they just completed a cell) while they are already participating in an auction, they log the update locally and communicate it to the other agents at the next scheduled synchronization event.

Ensuring that all agents are operating on consistent data is the first, and by far the most important, design challenge that we faced. If different agents in the system have different concepts regarding cell statuses, current bid values, or how far they are in an auction, the assignment solutions produced will be deeply flawed at best. To combat this and to make sure all agents possess the same concept of ground truth, we implemented three important methods for ensuring data consistency. They are *syncRounds*(), *cellStatusUpdate*(), and *bidStatusUpdate*().

3.3.1 Ensuring Data Consistency During Auctions in `AuctionSearch`

Multiple rounds of bidding are typically required when assigning agent-cell pairs if the agents are close together, as they often bid for the same cell. In subsequent rounds, losing agents can either increase their bids for their preferred cells or pursue different cells. Situations arise where some agents naturally get ahead of the others because they require less computation in a given cycle through the behavior loop. For example, an agent who has won their cell is only required to resubmit the same bid (their winning bid), while all losing agents are required to recalculate utilities and bid values.

If this process were allowed to proceed unchecked, the winning agents would start executing the next round of the auction before the losing agents entered that round, resulting in inconsistency issues regarding what each agent's current bid and targeted cell actually are. The *syncRounds*() method is implemented to make sure that all agents are executing the same round of the auction at the same time.

Formally described in Algorithm 6, the *syncRounds*() method combats data inconsistency by forcing agents who are ahead of others to wait to execute the next auction round until all other agents have caught up. This is accomplished by way of a corollary to the consensus minimum problem with a connected communications graph and no malfunctioning or misleading agents (i.e., no Byzantine failures) [47]. Any time an agent sends its cell statuses or bids to other agents, they attach the round number corresponding to the round within which they are operating. Agents receiving those messages maintain a set of reported round numbers. An agent cannot proceed to the next round until the consensus-obtained minimum equals the round it wants to execute. If an agent's round number equals the maximum of the set of agent round numbers, it must wait for the others to catch up.

Algorithm 6 Auction Round Synchronization with *syncRounds*()

round_tracker.add(*round_number*)
if *length*(*round_tracker*) == 0 **or** *length*(*round_tracker*) == 1 **then**
    return ***True***
else
    max_round_num ← *max*(*round_tracker*)
end if
if *length*(*round_tracker*) > 1 **then**
    if *round_number* == *max_round_num* **then**
        round_tracker.clear()
        auction_status_request()
        return ***False***
    else
        return ***True***
    end if
end if

Similar constructs were implemented for cell status updates, bid messaging, and auction complete messaging to help maintain synchronization by ensuring that agents are not permitted to proceed with the auction unless every participating agent has heard a current status from every other participating agent. Specifically, if an agent receives a bid from another agent which is tagged with a round number that does not match their own, the bid is rejected in order to enforce consistent round execution. Our implementation provides "previous request" functionality that prevents deadlock situations where one agent is trying to request information from other agents who are unwilling to send it, providing a way for agents to catch up.

Robotic systems intended to operate in the real world not only need to cope with various stages of execution, but they must also deal with lossy communications connections. The system must be robust to data loss during transmission. Each agent in our implementation keeps track of the number of agents executing `AuctionSearch` and checks whether it has heard from all other active agents during key synchronization steps such as during cell status updates or bidding. If an agent has not received messages from all other agents, it sends requests for the information. No agent will proceed with an auction round unless it has heard from all other participating agents. If some subset of agents have stopped executing `AuctionSearch`, their departure is detected by the remaining agents and they are excluded from future reporting requirements and are stripped of any cells they were responsible for prior to departure.

3.3.2 Generating Cell Utilities and Bids in `AuctionSearch`

Once all the agents have a common understanding of which cells are included in the auction and which are not, the next step is for each agent to determine for which cell they prefer to bid. To make this determination, each agent generates a utility value for each cell, choosing the one which nets the highest value for the agent. The accrued set of utility values is used to determine what bid the agent should place for its preferred cell. Below we detail how our implementation accomplishes this task.

Our implementation uses *generateCellUtilities*() to iterate through each cell and calculate the individual utility values. Equation 3.3, introduced in many forms in this thesis' references, shows how an agent calculates the utility for a given cell c [26], [27], [33].

$$utility_c = private\_value - utility\_cost - cell\_cost \tag{3.3}$$

$$utility\_cost = distance + size + remaining\_size. \tag{3.4}$$

The net utility associated with a particular cell, $utility_c$ in Equation 3.3, of a particular cell c can be described as the net value realized by the agent for owning it. The objective of each agent, then, is to maximize its own *utility* through selection of the highest-utility cell. In the same vein, *utility_cost* can be described as the cost incurred by an agent for owning a given cell. For the search problem of this thesis, this can be reasonably estimated as a function of the distance that the agent would be required to travel to complete the search of a particular cell (Equation 3.4). The components of this calculation are as follows:

1. *distance*: The Euclidean distance from a particular location to the closest starting waypoint within a given cell. If an agent is already searching a cell (i.e., the cell's status is *in_progress*), *distance* is calculated as the Euclidean distance from the last waypoint in the agent's search path to the best starting waypoint in the candidate cell.

The "best" starting waypoint is defined as the waypoint occupying the corner of the candidate cell which is nearest the agent. If an agent does not have any active cells, *distance* is the Euclidean distance from the agent's current location to the closest starting waypoint of the candidate cell.

2. *size*: The distance that the agent would be required to travel in completely searching the candidate cell. This value is a function of the size of the cell and the agent's sweep width (i.e., visibility of the ground at the search altitude). While this component remains constant in the basic environment (where cell sizes are uniform), it varies with cell size in complex environments containing obstacles non-uniformly shaped cells. The larger the size of the cell, the higher the cost of ownership since larger cells will generally take longer to search.
3. *remaining_size*: The distance that the agent is required to travel to complete the search of its current *in_progress* cell before transiting to the candidate cell. Similar to how *size* scales, *remaining_size* can be arbitrarily large in complex environments where non-uniformity can create arbitrarily large cells.

The above listed *utility_cost* components are depicted in Figure 3.8.

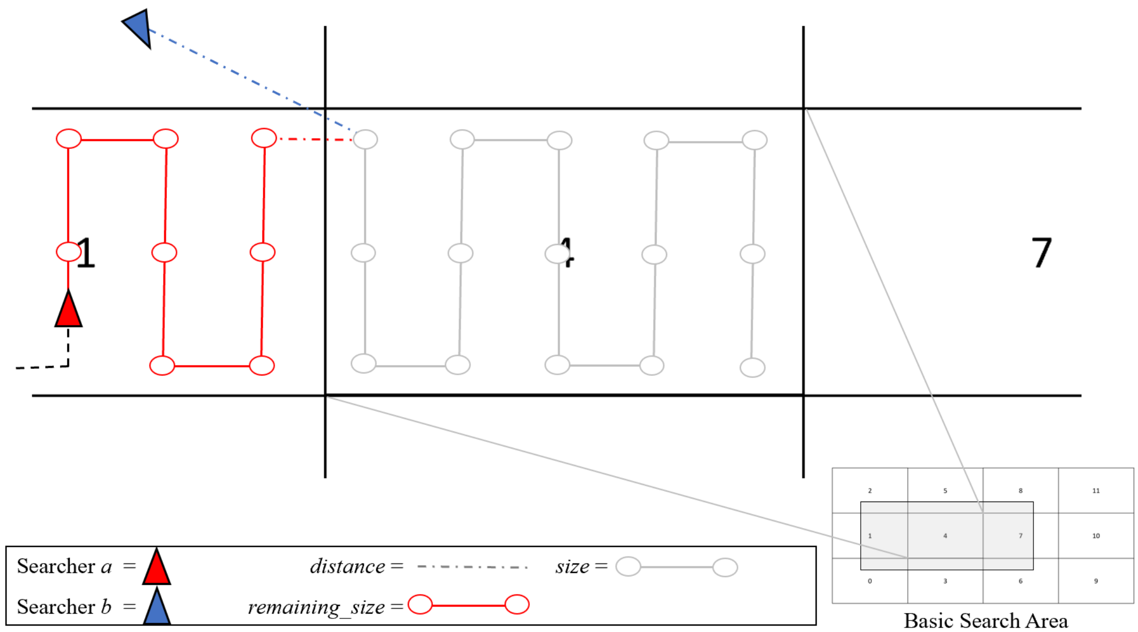

Figure 3.8. **Utility Cost Components.** The utility cost that an agent associates with a given cell is the combination of three components: *distance* to the cell, the *size* of the cell, and the *remaining_size* in an agent's current cell, if any.

3.3.3 Utility Function Variables

The components of the *utility_cost* of Equation 3.4 scale linearly as its individual components vary. The *size* and *remaining_size* values associated with a search area, for instance, vary in direct proportion to the cell's size (area). Similarly, the *distance* value is wholly dependent on vehicle locations and the Euclidean distance between individual search cells. These components evidently scale the entire system linearly because the utility cost components affect all agents equally and thus abstract away the specific agent locations at a particular time.

Given that our utility costs are a function of distance traveled, we modify each agent's bidding strategy by using this distance to derive their expected incurred costs for each candidate cell. Individualized bidding strategies ultimately allow each agent to maximize their utilization relative to the strengths and weaknesses of the other agents. The individual strategies we explore use per-robot speed and endurance to calculate cell utility as a function of time or energy required to search a cell. We then compare the results across the three search areas introduced in Section 3.2 with different mixes of highly capable (i.e., fast or high-endurance) and less capable (i.e., slow or low-endurance) agents. Below, we briefly describe how we use these values to calculate expected costs. We define the actual functions in Chapter 4.

1. *Speed*: Each agent is capable of a specific maximum transit speed to travel. We use individual speed to derive utility values that maximize system-wide efficiency by minimizing individual search times. We combine distance to travel with each agent's speed to compute the required search time for a particular cell. In order to avoid overly greedy results (e.g., fast agents dominating the entire search to the detriment of system-wide utilization) we provide an advantage bias to agent utility calculations that is inversely proportional to transit speed. As a result the utility costs for faster agents grow more slowly as cell sizes increase. The ultimate outcome that this dynamic achieves is that all agents prefer to search smaller, closer cells, but faster agents are less averse to searching larger, more distant cells.
2. *Endurance*: Each agent has a specific endurance threshold arising from its characteristics (e.g., battery capacity) and mission history (e.g., prior tasking). We use endurance to derive utility values that maximize system-wide efficiency by maximizing energy conservation. We combine distance to travel with each agent's endurance to estimate the required power usage, or effort, associated with a particular cell. Scaled in a manner similar to the speed utility, all agents prefer to search low-effort cells, but high-endurance agents are less averse to conducting more of the search workload than low-endurance agents.

3.3.4 Generating a Bid for a Cell

Once these elements are considered and the utility for each cell is generated, the maximum utility-producing cell is selected as the preferred cell and a bid for ownership is then computed using Equation 3.5 and communicated to the rest of the agents. Introduced in different forms in the auction algorithm literature [26], [27], [33], this equation computes a bid as a function of the highest-utility cell, the second-highest-utility cell, and a system-wide "minimum bid" value, (ϵ).

$$bid\_value_c = previous\_bid + highest\_utility - 2nd\_highest\_utility + \epsilon \qquad \epsilon > 0. \tag{3.5}$$

Our implementation's bid generation function serves two purposes. First, it provides agents the ability to compute new bids for each round that take previous results into account. Second, it provides the means to determine when the auction can be terminated. Below we discuss these major elements of bid generation in `AuctionSearch`.

For our implementation, we assume that search areas tend to contain more cells than there are agents to search them, given our cellular decomposition strategy. This implies that the majority of the auctions our implementation executes are instances of asymmetric assignment, where either the cells outnumber the agents (e.g., early in a search) or the agents outnumber the cells (e.g., at the end of a search). As such, the bidding and assignment phases of each round must account for this wide range of configurations.

In symmetric assignment, introduced in [27] and discussed in Chapter 2, the auction algorithm swaps n assignments among n agents and then measures whether they are within ϵ of their highest utility to determine whether to bid for a different cell or not. In our implementation, the bidding phase consists of agents bidding for their highest net-utility cell and checking whether any other agents submitted a higher bid for the same cell. If not, the round moves to the assignment phase and tentatively assigns the cell to the winning agent. If there was more than one bid for the same cell, the highest bid wins.

We manage bidding for cells by lumping agents into two bins. The first are those who won their favorite cell in the previous round and the second are those who did not. Figure 3.9 depicts the bid generation logic our implementation follows.

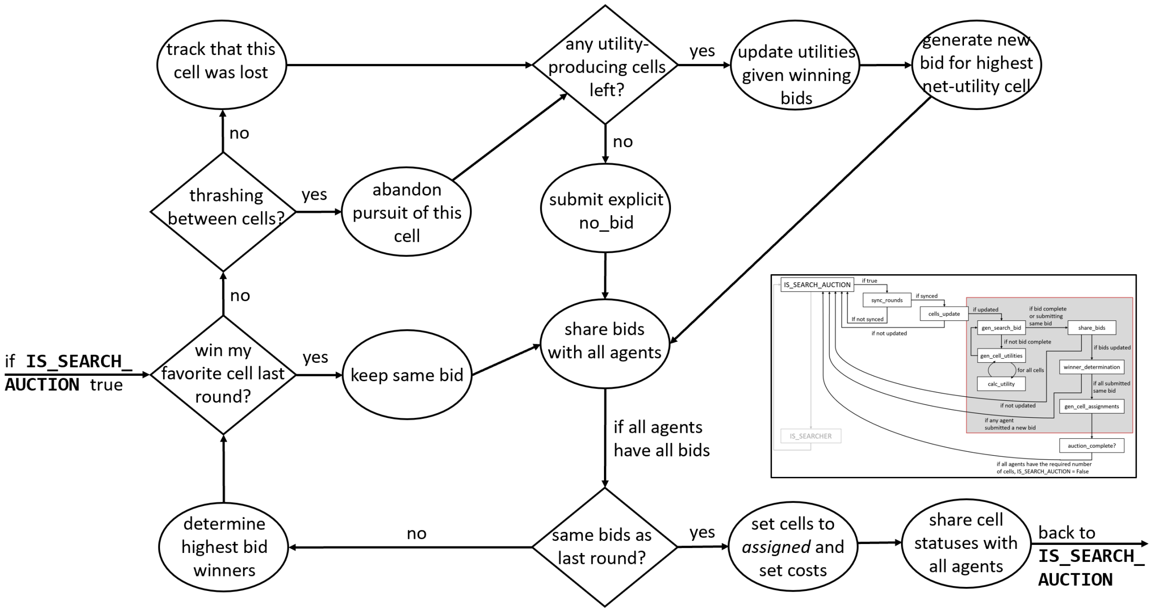

Figure 3.9. **Agent Bid Generation Logic.** This figure describes the logic that `AuctionSearch` agents follow when generating bids for cells. As fewer and fewer cells are available at the end of a search, only the agents with the highest utility values will win assignment while the others abandon cells and ultimately submit "no bids."

If agent a won its favorite cell in the previous round, it means that a's bid is the maximum value in the set of bids for a particular cell in a particular round. All agents who submitted winning bids in the previous round are directed to submit the same bid for the same cell again. When all participating agents have submitted the same bid twice for the same cell, and all agent-cell pairs are unique, the auction is closed and agents commit to their assignments.

Designing the algorithm to have agents submit their winning bid twice allows each agent to detect when all other agents are happy with their assignments, having selected net-utility-maximizing cells that are free of conflicts. This design is equivalent to having each agent send a specific message indicating satisfaction with the current assignment without the overhead of additional messaging. In the next subsection we describe how this behavior contributes to auction termination and cell assignment.

If agent a did not win its favorite cell, it means that two or more agents submitted bids for the same cell and a's bid is not the highest bid for its favorite cell. All losing agents are then required to compute new bids that take into account the winning bids from the previous round as well as increasing bids by at least ϵ. Agents compute new bids by logging the new tentative costs for cells and determining the impact of those costs on their net utility values. The losing agents then choose the cell with the highest associated net utility for their next bid.

When the number of agents is larger than the number of cells, some number of agents will necessarily fail to be assigned. In order to allow agents to abstain from bidding in a detectable way (e.g., more than simply not bidding, which could be ambiguously interpreted as a malfunction), non-bidding agents submit an explicit "no bid." Agents decide to abstain from bidding for a particular cell after they have lost the same cells multiple times back and forth, indicating thrashing between two or more cells with similar utility values. As agents decide to abstain from bidding as the auction proceeds, the set of cells eventually equals the set of winning agents, producing our one-to-one mapping. As such, as the number of cells continues to decrease at the end of a search, only the agents with the highest utility for the last remaining cells will win them.

3.3.5 Auction Round Winner Determination in `AuctionSearch`

Once each agent has generated and shared its bid for its preferred cell, the next step is to determine which bids are the highest and whether the agents are satisfied with their proposed assignments or not. Our implementation uses Algorithm 7 for this purpose.

The computational complexity of winner determination in the general case as implemented in Algorithm 7 is $O(cnm)$ where c is the complexity of our *consolidateBids*() step which checks for cell conflicts, organizes bids into dictionaries, and determines whether our termination criteria has been met. n is the number of agents bidding for cells and m is the number of bids per agent. An auction is terminated once each agent has submitted the same bid for the same, unique cell for two consecutive auction rounds.

Algorithm 7 Auction Round Winner Determination

all_bids ← dictionary of previous round's bids (if any)
inbound_bids ← list of other bids of form [[*searcher*, *cell*, *bid_value*] ...]
bid ← my bid of form [*searcher*, *cell*, *bid_value*]
same_bids ← ***True***
consolidateBids() and set *same_bids* ← ***False*** if bids are different from last round
if *same_bids* == ***True*** **then**
    Assign each *searcher* to *cell* for cost *bid_value* and set *cell*'s state to *assigned*
else
    winner ← my *searcher_id*
    highest_bid ← my *bid_value*
    for *i* = 0 to *length*(*all_bids*) **do**
        if other agent's *bid_value* > my *bid_value* for the same *cell* **then**
            highest_bid ← other agent's *bid_value*
            winner ← other agent's *searcher_id*
        end if
    end for
    if *winner* == my *searcher_id* **then**
        submit the same bid once again
        submit_same_bid ← ***True***
    end if
end if

Every bid from every agent is inspected in our consolidation step to check for termination criteria and conflicts, so this step requires n inspection operations. If the agents did not submit the same bids, our winner determination step conducts another n inspection operations to determine which agents won which cells. Therefore, in the worst case, Algorithm 7 requires at least n^2m inspection operations. Since our implementation follows the single-item auction paradigm, agents only bid for one cell at a time, fixing m at 1 and the computational complexity is therefore $O(n^2)$. The complexity of Algorithm 7 can be reduced to $O(n)$ if different auction termination criteria is used and the consolidation step omitted.

3.4 Conduct of an Area Search after Cell Assignment

In this section we describe the second major branch of execution in our implementation of the `AuctionSearch` of Figure 3.1. The `IS_SEARCHER` branch controls the actual search of assigned cells by assigned agents. Depicted in Figure 3.10, this Boolean-controlled branch is executed by each agent while there are still cells that reside in any state other than *complete*, as shown in Equation 2.13 and discussed in Figure 3.6. The *set_waypoint*() and *test_waypoint*() tasks execute every time-step and respectively set latitude, longitude, and altitude towards which the agent is to navigate and determine whether or not the current search waypoint has been reached.

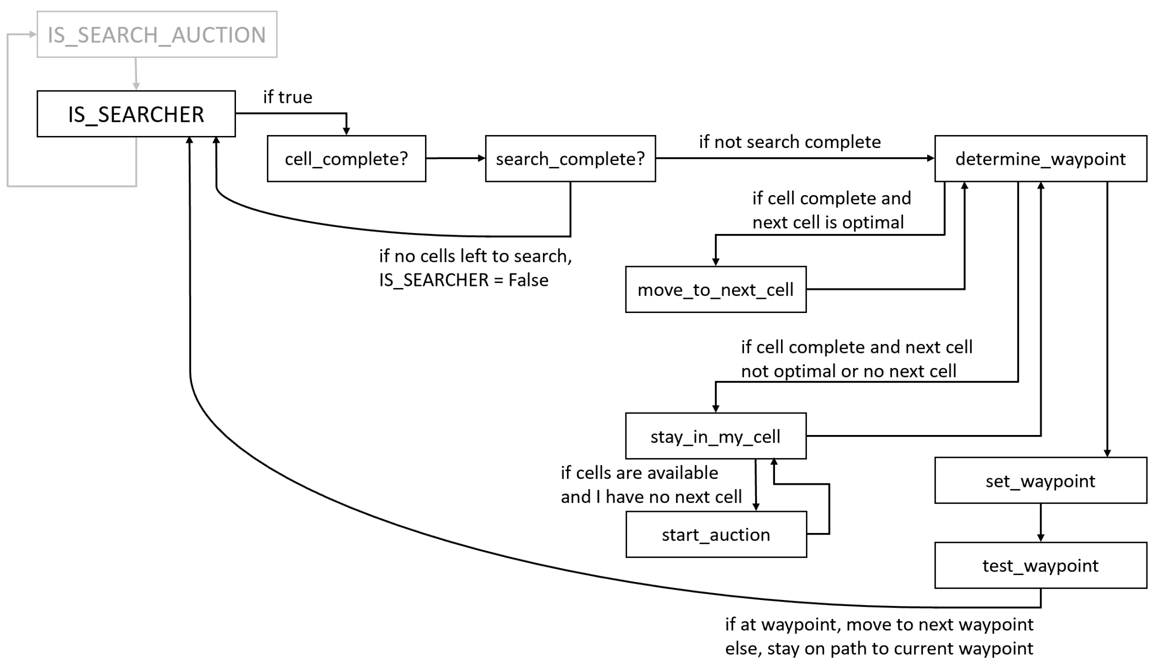

Figure 3.10. **Search Conduct in `AuctionSearch`.** This figure shows a deeper look at the search branch of control. `IS_SEARCHER` is *True* as long as there are cells left that are not *complete*. `IS_SEARCHER` becomes *False* when all cells are in the *complete* state, at which time the `AuctionSearch` algorithm terminates as well.

Other steps depicted in Figure 3.10 are executed as required based on the current search state. The current search state is a function of the collective statuses of the cells that make up the search area. The state change logic driving `AuctionSearch` is depicted in Figure 3.11.

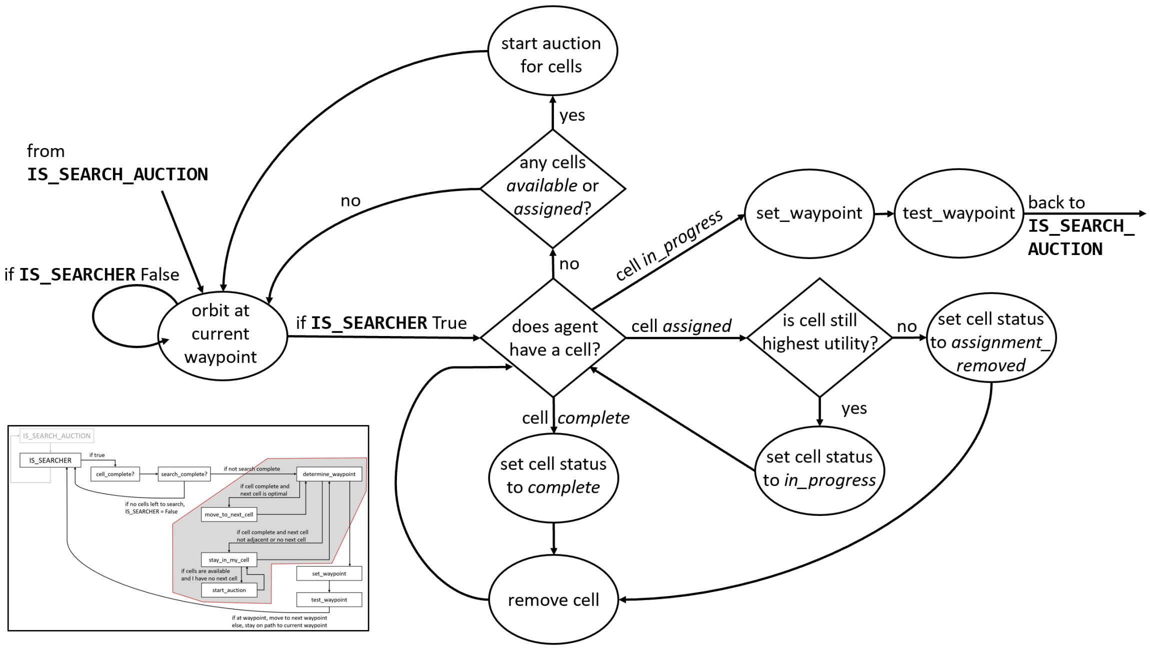

Figure 3.11. **Agent Cell Change Logic Diagram.** This figure show how cell state transitions are managed in `AuctionSearch`. Agents determine what action to take based on their current cell assignments, if any.

A more illustrative example is provided in Figure 3.12 to further clarify agent behavior based on current cell assignments. Referencing Figure 3.11 as well, Figure 3.12 shows how the auction process optimizes cell assignments based on cell utilities and current cell assignments. Agents are not obligated to search a cell unless it has set its status to *in_progress*, which only occurs if the agent has entered the cell and begun search. Therefore, *assigned* cells are available for auction in order to achieve higher system-wide utilization and efficiency. Searcher *a* decides to abandon its association with cell 5 in favor of cell 4 or 7, for example, due to its expected increase in utility for the association.

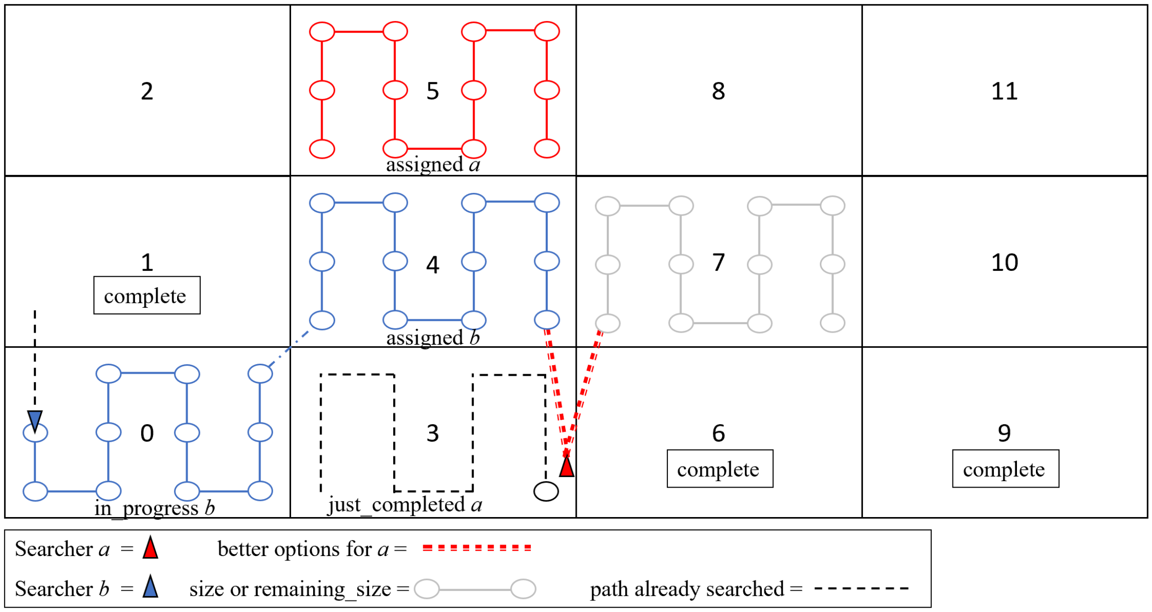

Figure 3.12. **Agent Cell Change Example.** This figure shows two agents and their current assignments at a particular time-step of an area search. We use cell search completion as a trigger criteria for a new auction. This provides the opportunity for agents to increase their per-agent utilization by reshuffling assignments to optimize system-wide utilization given current conditions. This ultimately results in a more efficient search.

3.5 Summary

In this chapter we introduced our auction-based area search implementation through which we experimented with various environmental and utility function considerations. We first described the `AuctionSearch` three search areas with which experiments were conducted. Each test search area provided a different scale and complexity to facilitate capture of realistic results for auction-based assignment of area search cells in challenging scenarios.

In this chapter we also described our algorithmic implementations for major aspects our use of auction algorithms to create cell-agent pairs for efficient execution of the area search. Covered topics included maintenance of consistency over the course of multiple rounds, decentralized winner and auction completion determination, and our adaptation of the auction algorithm utility and bid equations of [26], [27], [33] to identify locally optimal bids to maximize agent utility and avoid minimize system-wide cost. and our utilization of the cell statuses of [25] for cell state tracking.

Our agent utility functions include many different variables to allow us to explore the range of possible solutions that our implementation can produce. These variables ultimately allow us to observe the variance across a range of utility function implementations to measure their performance against area search benchmarks. In Chapter 4 we present the results of our simulated experimentation and live testing of our two utility functions across system sizes and search areas.

CHAPTER 4:
Analysis of Auction-Based Assignment in Area Search

In this chapter we discuss the auction-based assignment and performance of `AuctionSearch`. First, we discuss the interplay of cell utilities and their impact on agent bids. Then we introduce speed and endurance utility functions that influence agent bidding strategies. We then discuss our experimentation framework using those utility functions and our measures of performance.

Finally, we measure how well `AuctionSearch` uses auction-based assignment to complete area searches of varying complexity, and seek to draw conclusions about the use of auction algorithms in general for area search applications. We analyze `AuctionSearch` performance given various search areas, system sizes, and individual robot speed and endurance values. Throughout this chapter we discuss the design trade-offs required to implement `AuctionSearch` and the reasoning behind those decisions to inform future research in the area of autonomous decision.

4.1 Impact of Cell Utilities on Agent Bidding Strategies

Agent bids are a function of the difference between their favorite (highest utility) cell and their second favorite (second highest utility) cell, regardless of the utility function used to calculate those utilities. Figure 4.1 provides a descriptive exemplar depicting how utility costs impact cell utilities and agent bidding strategies.

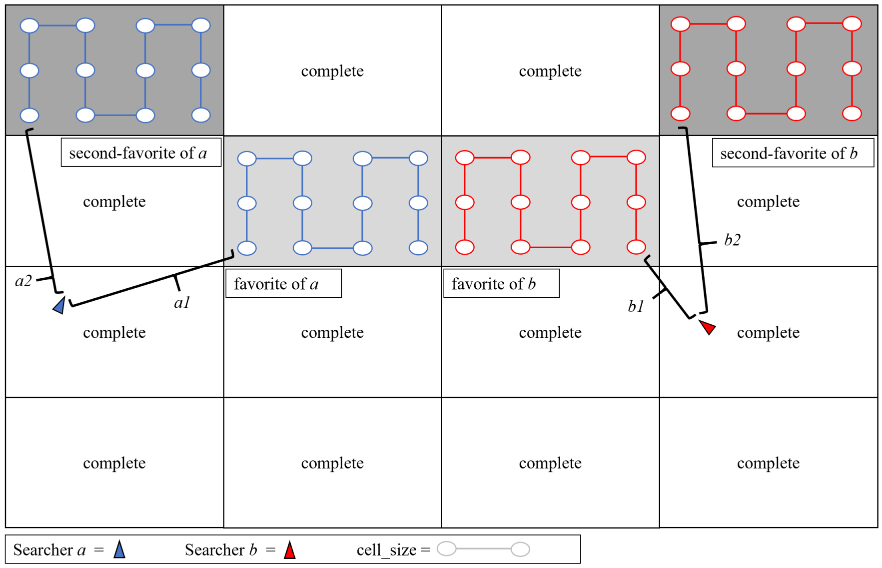

Figure 4.1. **Agent Bids Given Cell Utilities.** This figure shows two agents and the variables that effect bid creation. Agents' bids for their favorite cells reflect the interplay of favorite and second-favorite utility values. In this diagram, distances are $a1$, $a2$, $b1$, $b2$, $size$ is denoted by cell_size, and $remaining\_size$ equals 0.

An agent's bid for its favorite cell is dependent on how much worse its second favorite cell is because the bid is derived from the difference between the two utility values. In other words, as the difference between favorite and second favorite grows, the agent's bid for its favorite cell increases as well.

Different utility functions will generate different bidding strategies. As an initial example of a simple utility function's impact on agent bidding, Figure 4.1 depicts two agents that use minimum distance to determine their favorite cell. Searcher b favors its favorite cell because $b1 < b2$ with *size* and *remaining_size* utility cost components held constant. Further, $b1$ is much smaller than $b2$, making the difference between searcher b's favorite and second-favorite cells larger than for searcher a. Put differently, searcher b bids higher than searcher a because $(b2 - b1) > (a2 - a1)$. As a result, searcher b will submit a larger bid for its favorite cell than searcher a for its favorite.

Agent utility functions use this interplay of favorite and second-favorite cell utilities to manage the *utility dropoff*, referring to the rate of utility decrease as a function of *utility* *cost* from Section 3.3. Different utility functions can be used to increase or decrease agent *utility dropoff* rates on a per-robot basis, thus modifying individual bidding strategies using different robot characteristics such as speed or endurance.

4.1.1 Issues Associated with Distance-Dominated Utility Functions

Using distance alone to calculate utility values generates assignment solutions that tend to be overly greedy. By only considering distance in utility, each agent pursues its nearest cell exclusively, with *remaining_size* acting as the only major differentiating factor between individual *utility_costs*. In situations where multiple agents are collocated, this results in similar utility valuations and overlapping bids for the same cell across the system. When large numbers of agents come up with similar utility values for the same cells, multiple agents often identify the same highest-utility option. This creates longer than desired auction run times due to the large number of rounds required as agents seek more distant cells.

In Sections 4.2 and 4.3, we introduce two utility functions that take into account individual robot characteristics other than just distance to calculate utility. By making utility a function of individual capabilities, the multi-robot system is able to make better assignment decisions based on those capabilities and ultimately achieve higher system-wide utilization and efficiency.

4.2 Utility Function 1: Agent Utility as a Function of Speed

In this section we introduce our first utility function which uses speed to calculate cell utility values. By making utility a function of speed, agents use time-to-complete as the major differentiator between cells. We first define the function and then discuss the bidding strategy we expect from each agent given the impact of individual speed on cell utility.

4.2.1 Speed Utility Function Definition

Our first individualized utility function generates cell utility values as a function of speed. Our *speed_utility* function uses Equation 4.1 to calculate cell utility.

$$utility_s = value - \left(\frac{distance + size + remaining\_size}{speed}\right) - cell\_cost \qquad (4.1)$$

The first term of *utility*, which we call *value* in Equation 4.1, represents a large constant value from which the *utility cost* is taken. Our implementation treats all cells as equally valuable for the *value* variable so it remains constant, and negative utilities are allowed. The third term, *cell_cost*, is the additional cost a bidding agent is required to pay to take a cell that is already assigned to another agent.

The second term of *utility* in Equation 4.1 evaluates to the time required to complete transit and search of the prospective cell and the remainder of an *in_progress* cell, if any. This is also referred to as the *utility_cost* of owning the candidate cell. The faster the agent, the shorter the time it takes to complete a prospective cell. Our experiments use 15 and 23 meters per second (m/s) as the respective slow and fast speed values to provide measurable differences while staying within the flight tolerance of our Zephyr II airframes.

4.2.2 Expected System Behavior Given Speed-based Utility

For a faster agent, the difference between the highest-utility cell and second-highest-utility cell is less than for the slower agent. The slower agent, therefore, bids higher for the cell. This design encourages fast agents to let slow agents have closer cells because fast agents take less transit and search time for further cells than slow agents. By ensuring that slower agents have an advantage bidding for closer cells, system-wide utilization is maximized.

Faster agents suffer less utility dropoff as *utility_cost* increases due to their increased speed, so they are inherently capable of incurring such costs with less impact to system-wide efficiency than slower ones. Figure 4.2 shows how a slow agent and a fast agent calculate utilities for the same favorite cell. This figure depicts the notional bid values in Figure 4.3. Slow agents win their favorite cell in situations where fast and slow agents have similar, or identical, *utility_costs*.

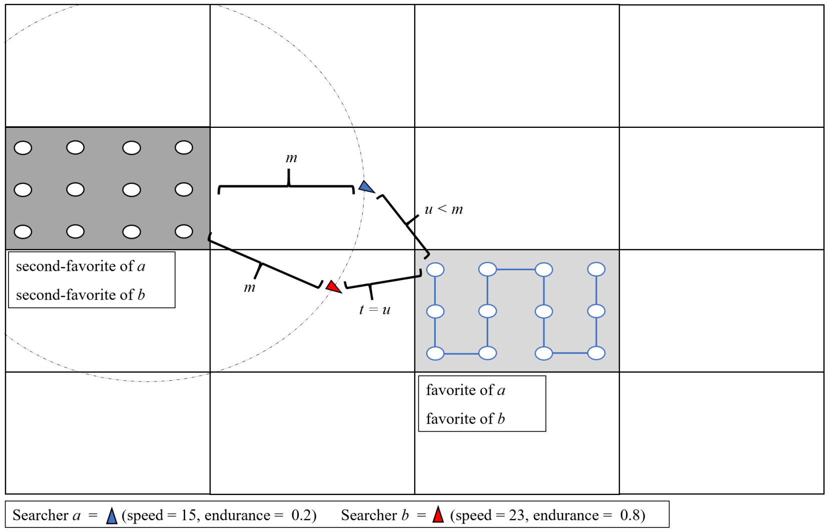

Figure 4.2. **Utility Scenario 1: Agents with the Same Utility Costs.** This figure shows how two agents with different speed or endurance values calculate bids for their favorite cell. In the depicted case, searcher a wins the cell because of its lower speed or endurance. For figure clarity, the *remaining_size* portion of *utility_cost* is 0.

The graph in Figure 4.3 shows an example of the relationship between a slow and fast agent bidding for their favorite cells based on speed utility calculated at increasing *utility_costs*.

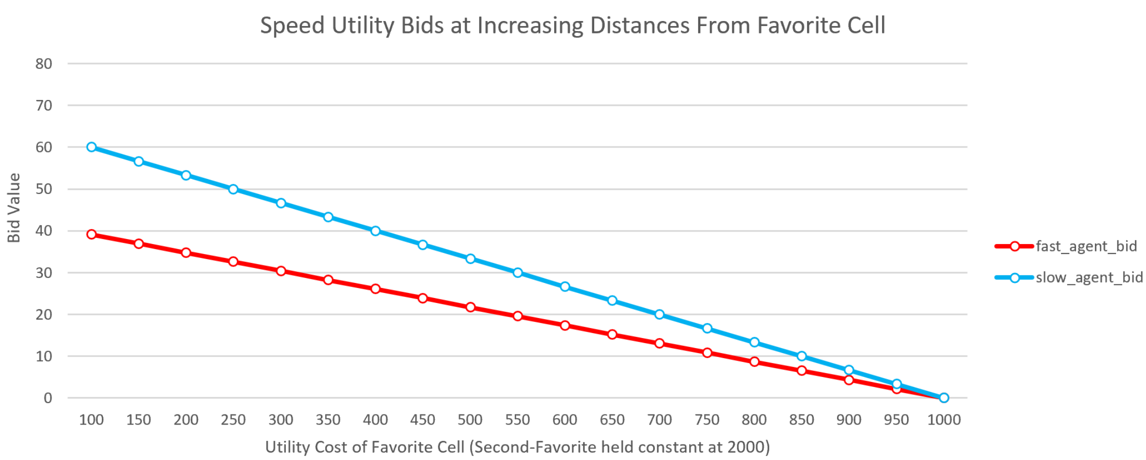

Figure 4.3. **Agent Bids at Increasing Speed Utility Costs.** This graph depicts how fast and slow agents' bids differ at increasing utility costs from their favorite cell. Agents bid less for their favorite cell the higher their utility cost. When utility costs between slow and fast agents are equal, slow agents outbid fast ones.

As expected, the higher the *utility_cost* associated with an agent's favorite cell, the less the agent is willing to bid for it. As shown in Figure 4.3, using our speed utility function affords slower agents an advantage over fast agents for their favorite cell (given equivalent second-favorite cell utility).

Agents rarely have the same utility costs and often have different utility dropoff values as a result. Slower agents have a steeper utility dropoff the higher their utility cost for their favorite cell is. Faster agents, however, experience a more gradual utility dropoff which means they are more inclined to stop pursuit of a closer, smaller cell than their slower counterparts. This relationship is depicted in Figure 4.4, showing how slower agents bid for, and win, lower utility cost cells.

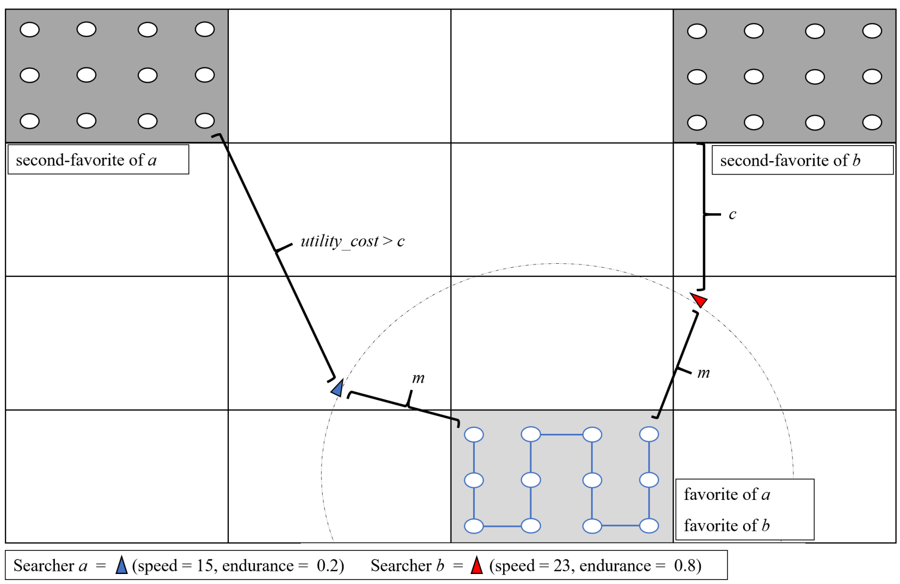

Figure 4.4. **Utility Scenario 2: Agents with Different Capabilities.** This figure shows how two agents with different speed or endurance values calculate bids for their favorite cell. Slow agents win their favorite cell versus fast agents when their utility dropoff is large. In the depicted case, searcher *a* wins the cell because its second-favorite cell achieves less utility than that of searcher *b*'s second-favorite. For figure clarity, the *remaining_size* portion of *utility_cost* is 0.

This utility relationship allows slow agents to be able to share in the workloads of very large and fragmented search environments where dramatic differences in favorite and second-favorite cell utilities can occur (e.g., our large-basic area introduced in Chapter 3). Figure 4.6 shows how slow and fast agents bid for the same cell when the slow agent has a much higher utility cost than the fast agent. A pictorial example of this scenario is presented in Figure 4.5.

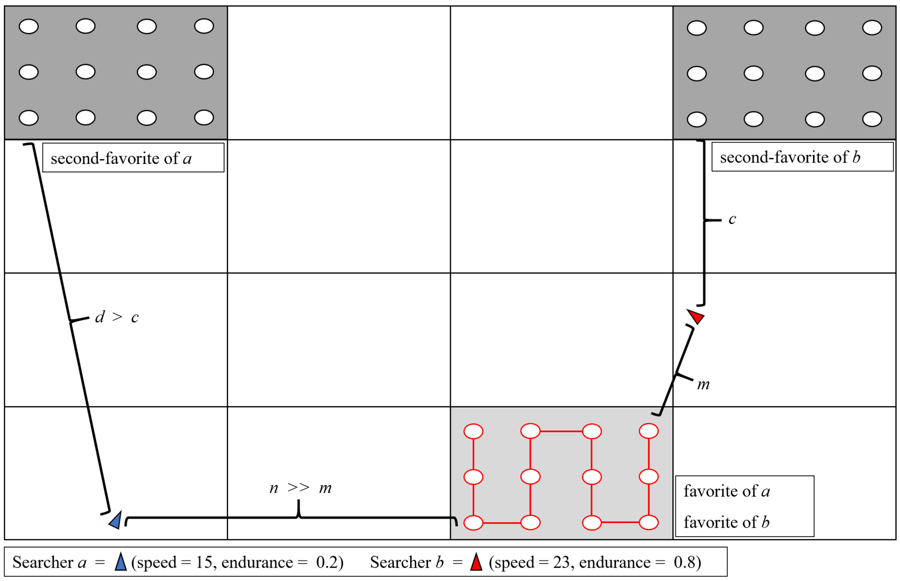

Figure 4.5. **Utility Scenario 3: Agents with Different Utility Costs for Same Cell.** This figure shows how two agents with different speed or endurance values calculate bids for the same favorite cell when the less-capable agent has a higher utility cost. In the depicted case, searcher b wins the cell because searcher a's utility cost is excessively high. For figure clarity, the *remaining_size* portion of *utility_cost* is 0.

The crossover point in Figure 4.6 is the point at which highly-capable agents outbid less-capable ones for the same cell, depicted in Figure 4.5. It is not advantageous for less-capable agents to win cells which incur much higher utility cost than for their highly-capable counterparts. Therefore, highly-capable agents win closer, smaller cells when less-capable agents are too far away or the cell is too large to benefit system-wide utilization or search time.

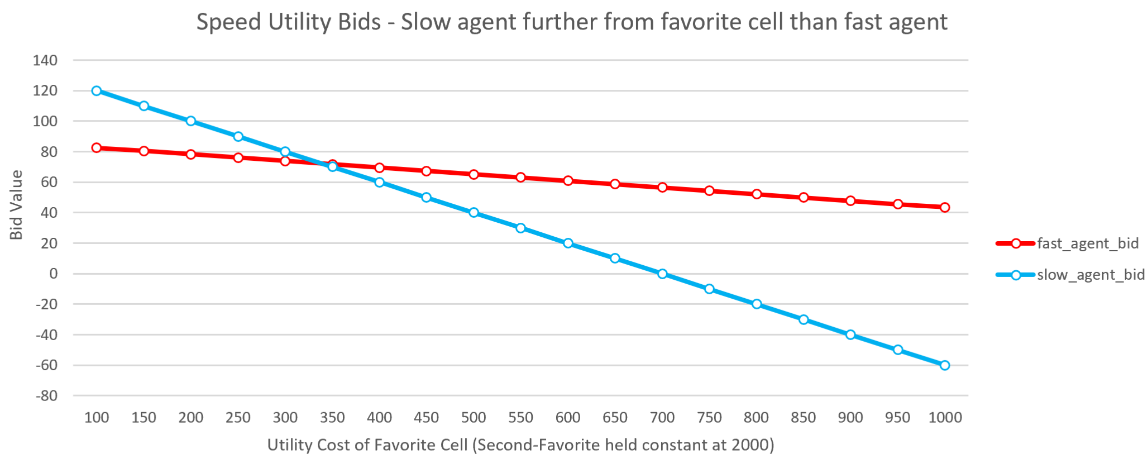

Figure 4.6. **Agent Bids as Slow Agent Costs Increase.** This graph depicts how fast and slow agents' bids differ as the slow agent's utility costs increase.

When the speed utility function is used in homogeneous systems (e.g., all fast or all slow agents), agents revert to greedy strategies due to their identical speeds. This maximizes individual utility in an effort to maximize system-wide utility like when using distance alone.

4.3 Utility Function 2: Agent Utility as a Function of Endurance

In this section we introduce our second utility function which uses endurance to calculate cell utility values. By making utility a function of endurance, agents use energy-to-complete, or "effort required," as the major differentiator between cells. As with the first utility function, We will define and then discuss the function's impact on agent bidding strategies.

4.3.1 Endurance Utility Function Definition

Our second individualized utility function generates cell utility values as a function of agent endurance. Using endurance makes the size and distance to each cell the dominant differentiator between agent utility values for the same cell. Our *endurance_utility* function uses Equation 4.2 to calculate cell utility.

$$utility_e = value - \left(\frac{distance + size + remaining\_size}{10 \times endurance}\right) - cell\_cost \qquad (4.2)$$
$$0 < endurance < 1$$

The *utility* in Equation 4.2 represents the effort required to complete the search of the candidate cell. Agents with lower endurance incur higher utility costs. We use 0.2 and 0.8 units as our low and high endurance values. The *value* and *cell_cost* terms are the same as in Equation 4.1.

4.3.2 Expected System Behavior Given Endurance-based Utility

The goal of our endurance-based utility function is to allow low-endurance agents to bid higher for smaller and closer cells than their higher-endurance counterparts. The more endurance an agent has, the more time and energy it can devote to transit and search, enabling it to be less averse to searching more distant or larger cells.

This design encourages high-endurance agents in much the same way our speed utility function encourages fast agents. High endurance agents allow lower-endurance agents to to take smaller, closer cells in an effort to get the maximum utilization possible from them. High-endurance agents suffer less utility dropoff as *utility_cost* increases, so are more inclined to find higher-cost cells acceptable than their low-endurance counterparts.

The graph in Figure 4.7 shows how low and high-endurance agents bid as utility costs for their favorite cells increase.

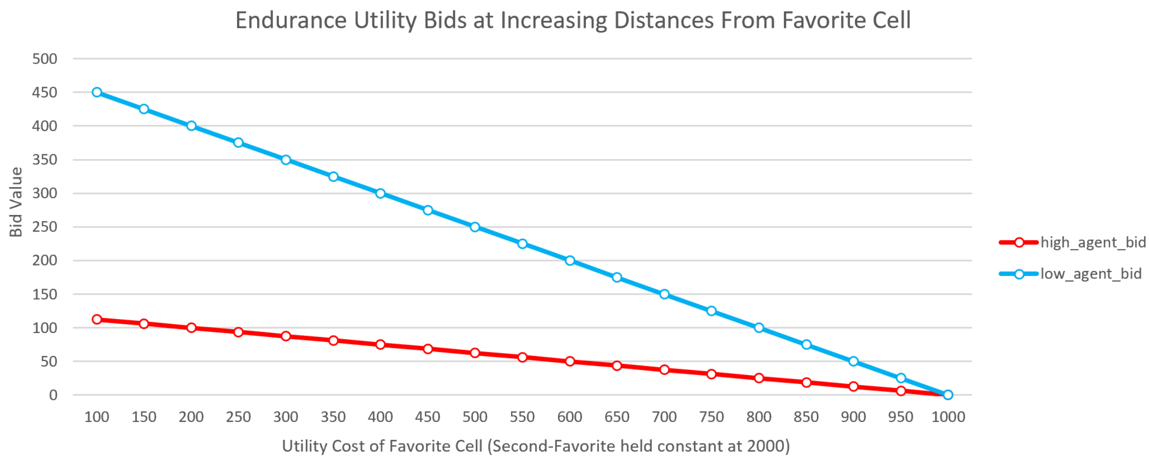

Figure 4.7. **Agent Bids as Endurance Utility Cost Increases.** This graph shows how agents' bids are impacted by utility costs with the endurance utility function. Low-endurance agents have a steeper utility dropoff and therefore outbid high-endurance agents for their favorite cells.

When low and high-endurance agents are considering cells with similar *distance*, *size*, and *remaining_size* values, low-endurance agents have a steeper utility dropoff from their favorite to the second-favorite cells. This is indicated in Figure 4.7 by the difference in slope between the two agents. As both high and low-endurance agents' utility costs increase, they both bid less due to the higher effort required. The higher-endurance agent, however, incurs less utility dropoff than the low-endurance agent because the difference between its favorite and second-favorite cells is small. This results in high-endurance agents allowing their low-endurance counterparts to win cells when utility costs are small. The high-endurance agents then pursue cells with larger utility costs.

Figure 4.8 shows what happens when low-endurance agents incur higher utility costs than high-endurance agents (e.g., low-endurance agent is at an increased distance from the cell than the high-endurance agent). High-endurance agents eventually outbid their low-endurance counterparts because the low-endurance agents' advantage erodes as utility cost increases. This causes high-endurance agents to bid for, and win, nearby and small cells if low-endurance agents are too far away to outbid them.

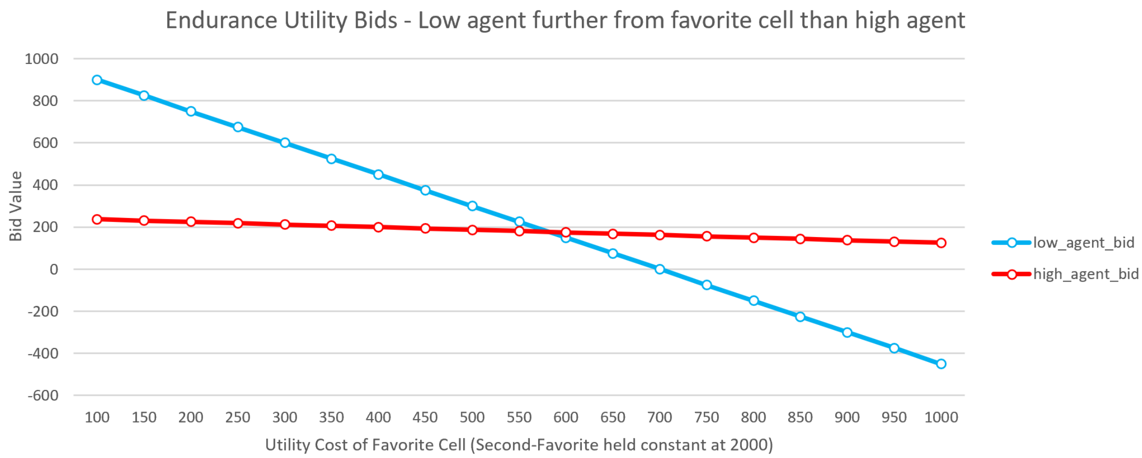

Figure 4.8. **Agent Bids as Low-Endurance Agent Cost Increases.** This graph shows how agents' bids differ as the low-endurance agent's utility costs increase. The result of, and motivation for, this behavior is the same as for the speed utility function described in Figure 4.5. Eventually, the high-endurance agent outbids the low-endurance agent.

4.4 `AuctionSearch` Experiment Setup and Performance Measurement

In this section we describe our experiment methodology. We now pivot from the introduction of our implementation and its utility functions to the measurement of its performance as an auction-based, area search driver. We first define our measures of performance that serve as the basis of our analysis, and then we introduce the framework that we tested `AuctionSearch` in to capture those measures.

4.4.1 Metrics for `AuctionSearch` Success

Here we list the metrics captured during simulated and live demonstrations of `AuctionSearch`. We use these metrics to quantify the variance between different runs with different configurations. Ultimately, these metrics allow us draw conclusions about system efficiency and identify areas where efficiency could be improved using alternative auction implementations.

1. *area search completion time*: The amount of time that a multi-robot system requires

to search a specific area running `AuctionSearch`. This value is compared to the amount of time the perfect search takes with the same search area, number, and type of robots.

2. *number of auctions*: The number of auctions required to assign all cells in a given search area with a given number of agents. The number of auctions required is a function of the number of cells, agents, and the maximum number of cells allowed to be won per agent per auction.
3. *average auction time*: The average amount of time agents spend in each auction. This metric is a function of the number and length of the rounds per auction. Conclusions about auction efficiency are impacted by how long each agent takes to complete each round, while conclusions about agent bidding strategies are impacted by the number of times agents sought the same cells as discussed in Section 4.1.
4. *average rounds per auction*: The number of rounds required for each auction divided by the number of auctions. This metric is used to observe agent bidding strategies and whether system-wide efficiency is impacted when agents are tightly bunched together.
5. *average round times per auction*: The seconds per round divided by the number of rounds per auction. This metric is used to observe the efficiency of the bidding phase of our auctions.
6. *per-robot contribution*: The percentage of the total search area that a particular robot completes. This metric is used to determine if subsets of agents conducted the majority of the search or if the workload was relatively dispersed system-wide. This metric, combined with *per-robot utilization*, allows us to draw conclusions about specific robot characteristics and their impact on the system's search completion.
7. *per-robot utilization*: The ratio of time an agent is actively engaged in search related tasks. We derive utilization for robot i searching cell j by Equation 4.3 where u = *utilization*, r = *run_time*, t = *transit_time*, and l = *loiter_time*:

$$u_{ij} = r_{ij} - t_{ij} - l_{ij} \tag{4.3}$$

We further define *transit_time* as the time robot i spends moving to cell j and *search_time* as the time i spends searching j (j is *in_progress*). We define *loiter_time* as the time i spends waiting for a cell assignment or for the end of the search, whichever comes first. More specifically, *loiter_time* = *total_time* − *transit_time* −

search_time. We analyze the different components of Equation 4.3 with regard to search area complexity, system size, and individual robot characteristics. We use this analysis to derive system-wide efficiency as a function of individual robot efficiency and to draw conclusions about the system's assignment decisions.

4.4.2 Experimentation and Data Collection

Our experimentation with `AuctionSearch` consisted of live and simulated flights of between three and 10 ARSENL-owned Zephyr II UAVs per run. Our live experimentation was conducted at McMillan Airfield, Camp Roberts, CA, and our simulated experimentation was conducted in the ARSENL SITL simulation environment. In total, we ran `AuctionSearch` 326 times in simulation and live-flight against our three search areas, two utility functions, and various system sizes. Figure 4.9 shows the breakdown of those runs per search area for each utility function.

AuctionSearch Experimentation: Total Runs				
	Speed Utility		Endurance Utility	
Large Basic Area	Number of Robots	Number of Runs	Number of Robots	Number of Runs
	6	10	6	10
	7	10	7	10
	8	10	8	10
	9	10	9	10
	10	10	10	10
	total speed runs in Large Basic Area:	**50**	**total endurance runs in Large Basic Area:**	**50**
Complex Area	Number of Robots	Number of Runs	Number of Robots	Number of Runs
	6	10	6	10
	7	10	7	10
	8	10	8	10
	9	10	9	10
	10	10	10	10
	total speed runs in Complex Area:	**50**	**total endurance runs in Complex Area:**	**50**
Basic Area	Number of Robots	Number of Runs	Number of Robots	Number of Runs
	3	10	3	10
	4	10	4	10
	5	10	5	10
	6	10	6	10
	7	10	7	10
	8	10	8	10
	total speed runs in Basic Area:	**60**	**total endurance runs in Basic Area:**	**60**
total number of runs:		**160**		**160**

Figure 4.9. **Total Number of Simulation Runs.** This table shows the various configurations observed in SITL. Even numbered robots are fast/high-endurance and odd numbered robots are slow/low-endurance. Agent numbering begins at 1. The additional six runs referenced above were live-flight validation tests.

In addition to varying the system sizes and the utility functions in each search area, we varied the mix of fast or high-endurance and slow or low-endurance for each system and we varied the agent start locations. We modified the mix of agents by selecting even-numbered agents in each run to be the fast or high-endurance agents, with their odd-numbered counterparts assigned as slow or low-endurance. This allowed us to observe the effects of the utility functions on the overall search.

We varied the agent start locations by either starting in a cluster (i.e., all n participating agents orbiting at roughly 140 meters away from the same waypoint) or by starting each robot in a pseudo-randomly chosen location inside the search area. By varying the start location, we are able to observe the effects of clustered-agent competition on overall search performance versus a more dispersed, less competitive start.

We purposefully chose to test `AuctionSearch` against a wide range of areas, system sizes, utility functions, and robot dispersion levels instead of testing against a single configuration exhaustively. We chose this in order to observe the results of auction-based assignment across a variety of configurations. Each run of `AuctionSearch` generates different assignment solutions and run times, even from tightly controlled starting configurations, due to the stochastic nature of agent cell completion patterns and auction initiation trigger times.

In addition to measuring performance across different system sizes, we also measure `AuctionSearch` against the perfect search as was done in [22] to provide a basis of comparison that is independent of our implementation. We use the same equation for the perfect search as was used in [22], rewritten here for convenience, with $T = search\_time$, $A = area$, $V = velocity$, $W = sweep\_width$, and $N = number\_of\_agents$:

$$T = \frac{A}{VWN} \tag{4.4}$$

For our perfect search calculations, we defined $W = 75m$, and $3 \leq N \leq 10$. V is equal to $15m/s$ for measurement against our endurance function, and V is equal to the average system-wide speed for our speed function. While this measurement is more useful for our speed utility function's performance, we include it for the endurance function as well for completeness. High-endurance agent utilization is more useful in measuring our endurance utility function's performance, as higher utilization indicates more system-wide efficient use of available energy.

4.5 `AuctionSearch` Simulation Performance in Various Search Areas

In this section we discuss the performance of our area search application in the SITL simulation environment. We used simulation for two reasons. First, physics-based simulation allows us to gather far more iterations of data than live-flight would allow. Second, using simulation allowed us to run the application at scale. Airspace restrictions and safety concerns limited the size of the area for which we could gather live-flight data. Our simulated large and complex areas allowed us to observe system reactions to realistic, large search areas with many cells to auction. Figure 4.10 shows the output of a typical experiment with the SITL simulation environment.

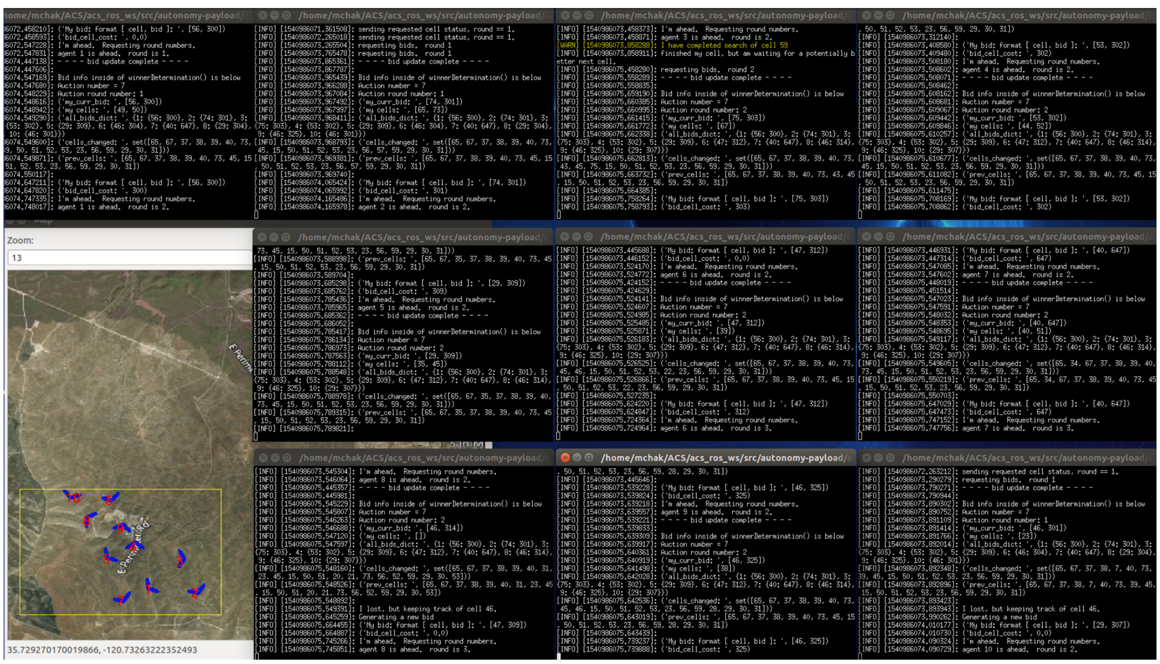

Figure 4.10. **Screen-shot of a 10-Robot Run in SITL Simulation.** This figure shows the user interface to the SITL simulation environment. Each terminal window displays robot-specific state information while running the behavior, while the overhead view in the lower left shows each robot's location in the search area.

Each subsection below describes the results from our experimentation in each of the three search areas utilized. We organize our results below by search area and by utility function, with speed utility followed by endurance utility. First, we present the overall outcomes for each run, then we graph the average results per-robot, per-run. We present results at the per-robot level because our analysis in Section 5.1 is dual-focused on per-robot contribution to the area search as well as the multi-robot system's performance overall. We focus at both levels because individual performance given speed and endurance impacts the system-wide performance of auction-based assignment and search completion. In the next section we analyze the results, draw conclusions, and explain our findings about auction-based area search.

4.5.1 Results from Area Search in the Large Area

In this subsection we present `AuctionSearch` results in the large area. This environment is the same size as the complex area, but contains 5 times the number of cells, all of uniform size. The goal of testing our implementation in this environment is to observe how systems react when large numbers of cells require search. In this area, *distance* plays a dominant role in utility costs as cell sizes are the same. Further, agent decisions have a larger impact on overall search efficiency in the large area because of the prevalence of orphan cells (i.e., those cells which have been left behind as the search progresses) which must be cleaned up as the search draws to a close. While these factors impact overall runtime in the large area, auction statistics and overall division of work across agent capabilities is fairly stable as system size increases. Figure 4.11 shows auction statistics for the large area across system sizes for both utility functions.

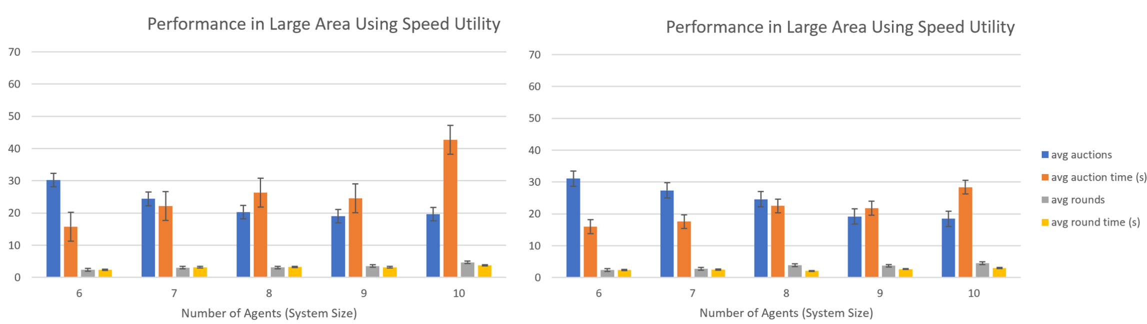

Figure 4.11. **Auction Performance in Large Area.** This figure shows how our implementation performs in the large area with regard to average auction and round counts and duration. The number of auctions, rounds, and their duration (measured in seconds) indicate how much internal deliberation the system requires to complete the search. As system size increases, auction counts decrease due to the increased number of cells assigned per auction.

Average auction durations, round counts, and round durations are all relatively low compared to the complex and basic areas. This is attributed to the fact that the cells outnumber the agents for the vast majority of the search. Agents have many cells with similar utility values to choose from, so the incurred utility dropoff is lower than in the complex area and suitable replacements are pursued. This results in lower relative auction times as fewer rounds are required to achieve unique cell-agent pairs.

The number of auctions required to complete the search in the large area generally decreases linearly as system size increases. When more agents are conducting the search, more work is being assigned per auction, until such time that the agents outnumber the cells. This result is echoed in the overall runtimes, shown in Figure 4.12, as overall runtime generally decreases with increased system size. The solid lines correspond to our implementation's average overall runtime for both utility functions. These include the time spent orbiting (i.e., *loiter time*) waiting for assignments if agents are participating in auctions with no *in_progress* cell. The dashed lines correspond to our implementation's performance when *loiter time* is removed, which we call *worktime*.

We compare runtime and worktime to the perfect search because our implementation incurs high runtimes due to our auction-trigger criteria being cell completion. This strategy often causes agents to orbit in place while conducting auctions if they do not have a follow-on cell, increasing overall runtimes. Comparing worktime to the perfect search is a more appropriate measure of the algorithm than our overall runtime since worktime removes this implementation-specific factor. The graph of the perfect search runtime at each system size is represented by the dotted line for comparison as well.

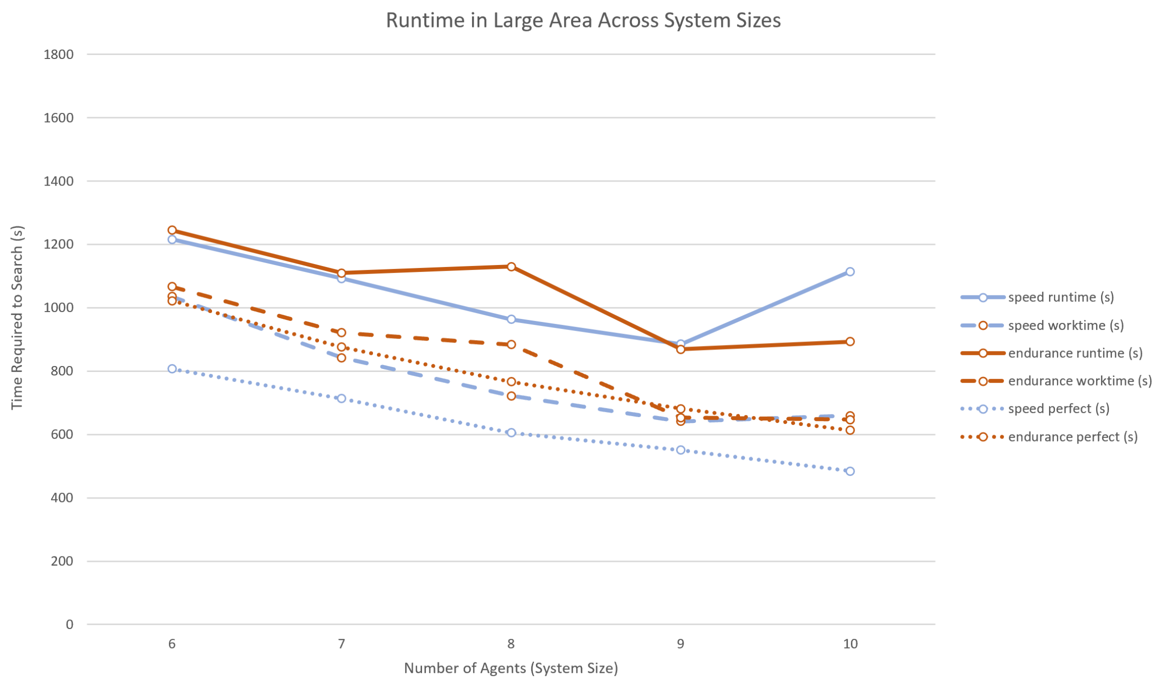

Figure 4.12. **`AuctionSearch` Runtimes and Worktimes in Large Area.** This figure shows the average runtimes and worktimes for the large area across system sizes. The time to search generally decreases as more agents are involved in the search.

Search of the large area benefits from increased system size more than either of the other two test environments because of the large number of cells within it. The more cells there are, the less competition there is among agents for assignments, so assignments are generated more quickly than in the basic or complex areas.

Another important aspect of area search is each agent's contribution to the overall search. We captured loiter time, transit time, and percentage searched for each agent at each system size to measure each agent's contribution given their specific characteristics (i.e., fast or slow, high-endurance or low). Figure 4.13 shows the percentage split of work completed by fast or slow agents (using the speed utility function) and by high and low-endurance agents (using the endurance utility function) in the large area across system sizes.

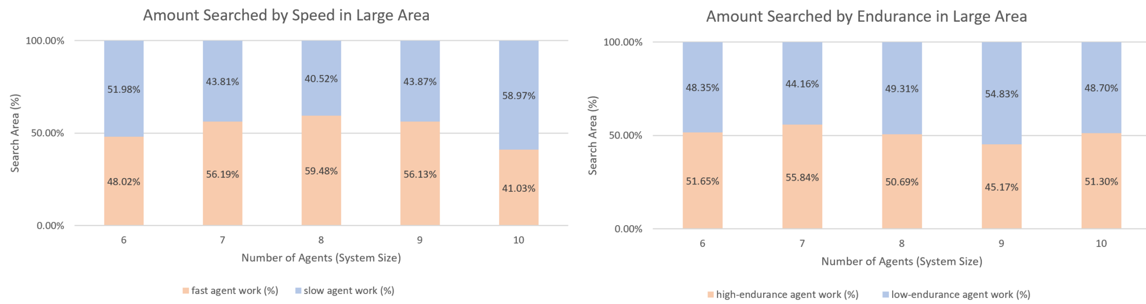

Figure 4.13. **`AuctionSearch` Division of Work in Large Area.** This figure shows the percentage of work completed by highly capable (i.e., fast or high-endurance) versus less capable (i.e., slow or low-endurance) agents across all tested system sizes in the large area. An equitable distribution of work across system sizes is demonstrated, which is a result of the search area consisting of uniform and relatively small cells that all agents achieve high utility for searching.

The division of labor in the large area was roughly equal across all tested system sizes. This is achieved because the search area consists entirely of small, uniform cells. While the cells outnumber the agents, both highly capable and less capable agents are able to find small, and relatively close, cells which achieve high utility. This causes all agents to contribute more or less equally to system-wide utilization, maximizing efficiency. This equitable distribution of labor breaks down once the agents begin to out number the cells, such as toward the end of a given search. This breakdown is most pronounced when assignment patterns have left distant orphan cells, which require agents to consider large distances in their utility cost calculations.

4.5.2 Results from Area Search in the Complex Area

In this subsection we present `AuctionSearch` results in the complex area. This environment is 16 times larger than the basic area. It is also more complex than the large area due to its various cell sizes. In this search area, *size* and *remaining_size* have the largest impact on agent bidding strategies given our utility functions. The large variance in cell utility introduced by these utility cost elements enables highly capable agents (i.e., those possessing high speed or endurance) to contribute more to the search than their less capable counterparts. Figure 4.14 shows auction performance in the complex area across system sizes for both utility functions.

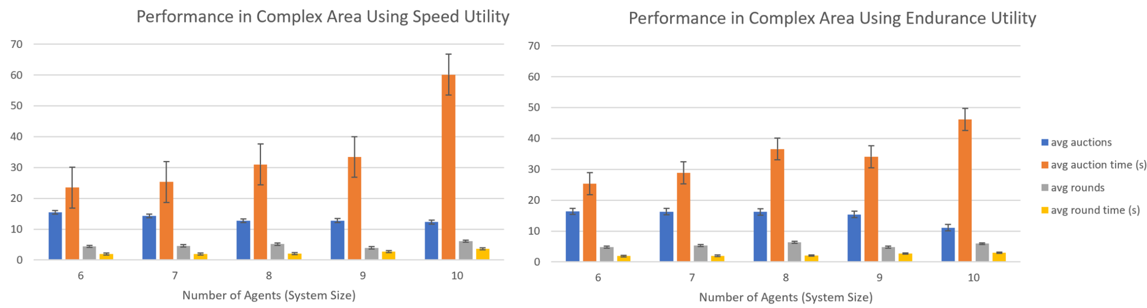

Figure 4.14. **Auction Performance in Complex Area.** This figure shows auction performance in the complex area with speed utility (left) and endurance utility (right). Average auction duration is higher in the complex area than in other test areas due to agents adamantly wanting smaller cells and entering bidding wars for them to avoid being assigned massive cells.

As expected, agents consistently pursued smaller cells instead of larger cells because of the large disparity in utility cost. The higher average number of auctions (indicated by the blue bars in Figure 4.14) is attributed to under-utilized agents initiating auctions for cells that are assigned to, but not yet set to in-progress by, other agents. This begins to occur once the agents outnumber the cells during the search, and the under-utilized agents (i.e., those with no assignment) start an auction after a predefined amount of time. Under-utilized agents continue to initiate auctions at predefined intervals as long as cells remain in either the *assigned* or the *available* state in an effort to ensure that only agents with the highest utility for a given cell end up searching it. Due to the large distances between cells in the complex area, agents assigned cells can be in transit for extended periods of time, enduring numerous auctions during which those agents must defend their assignment to the given cell against the under-utilized agents. This ultimately leads to higher average auction counts in the complex area, but ensures that highest utility is achieved.

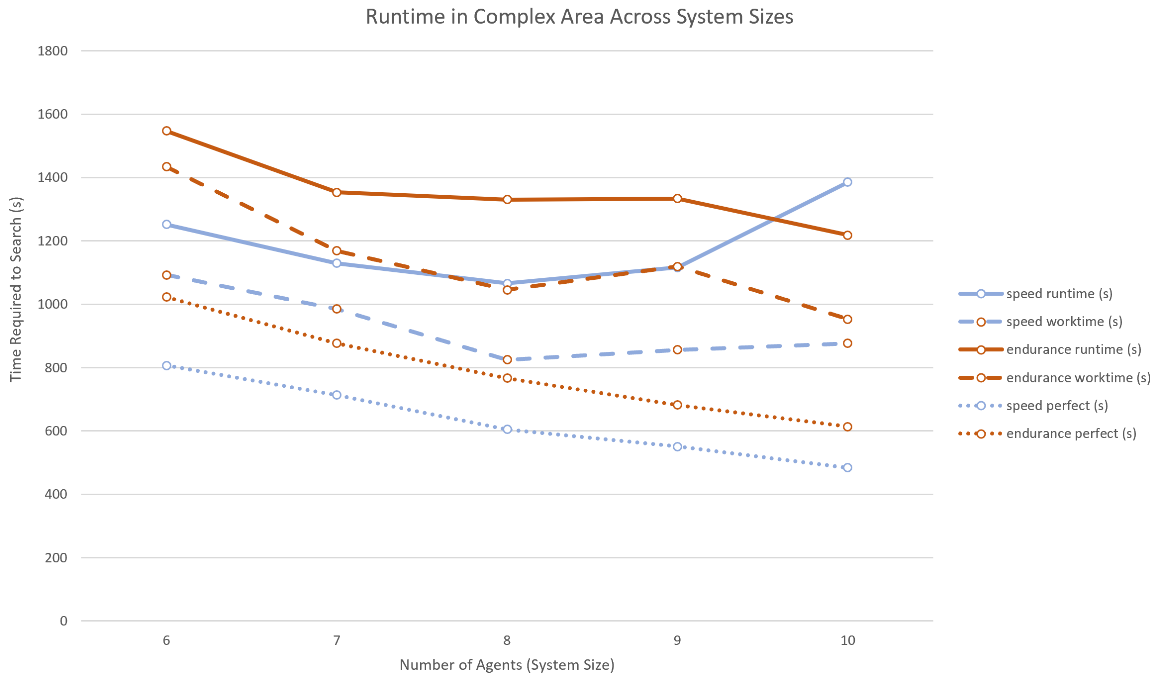

Figure 4.15. **`AuctionSearch` Runtimes and Worktimes in Complex Area.** This figure shows the overall runtimes and worktimes of our implementation in the complex area. Runtimes are longer than in the perfect search across both utility functions and across all system sizes due to agent deliberation and bidding wars for limited numbers of attractive cells. In the complex area, small cells are generally more attractive than larger cells due to their greatly reduced utility costs.

Figure 4.15 shows the runtimes and worktimes of `AuctionSearch` in the complex area across system sizes for both utility functions and as compared to the perfect search. Runtimes are higher than in the perfect search due to the deliberation required to achieve assignment and our implementation decision to start auctions for follow-on assignments only after a cell is set to *complete*. Worktime (i.e., our runtime with time spent loitering removed) trends lower as system size increases, mapping closer to the perfect search than our overall runtimes.

Average runtimes are particularly high as system size increases across both utility functions for two reasons. First, agents continually compete for cells back and forth as the search draws to a close, with each agent competing for fewer and fewer cells until they eventually hit the explicit “no bid” criteria discussed in Chapter 3. Once all of the small cells are already in-progress or complete, agents experience very high rates of utility dropoff from their favorite to their second-favorite cells in the complex area due to largely variant cell sizes. This causes all agents to pursue their favorite cell more per auction than if they had less-costly alternatives to fall back on. The second reason for large average run times is the impact of the requirement for system-wide consensus on shared data such as the cell states and bid values for an increasing number of agents before auctions are permitted to proceed.

Runtimes and worktimes are generally longer when using the endurance function versus the speed function because the high-endurance agents are required to carry a majority of the workload due to the massive utility costs associated with the majority of cells in the complex area. This resulted in the low-endurance agents completing all of the smaller cells while the high-endurance agents were generally responsible for searching all of the larger cells, requiring large amounts of time to search. This result is also apparent in the percentages searched given speed and endurance in Figure 4.16.

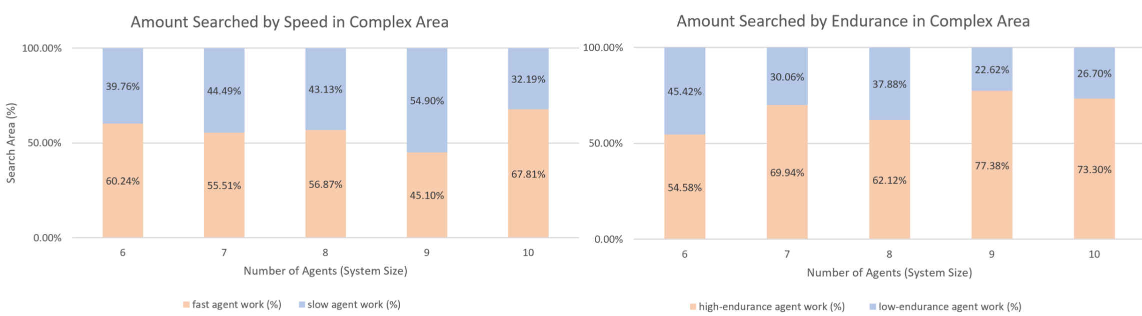

Figure 4.16. **`AuctionSearch` Division of Work in Complex Area.** This figure shows the division of labor between fast and slow agents as well as between high and low endurance agents across various system sizes. In general, fast and high-endurance agents conduct the majority of the search compared to their slow and low-endurance counterparts.

Figure 4.16 shows how fast and high-endurance agents generally dominate the search due to their willingness to search larger and more distant cells. The complex area is made up of mostly large cells, so agents searching those cells contribute a larger amount to search utilization than agents searching small cells. Additionally, as system size increases, the division of labor tips toward fast and high-endurance agents. This is due to more fast and high-endurance agents being available in larger systems to take the larger cells (given that even-numbered agents were set to high capability) which leads to those highly capable agents taking more of the work as more of them participate.

4.5.3 Results from Area Search in the Basic Area

In this subsection we present `AuctionSearch` results in the basic area. This environment is the same size and structure of our live-fly area, and represents the smallest and most constrained environment in which we tested our implementation. As such, the relative number of auctions and the overall runtime of searches in this area were small compared to the complex and large search areas. The auction results in the basic area are presented in Figure 4.17.

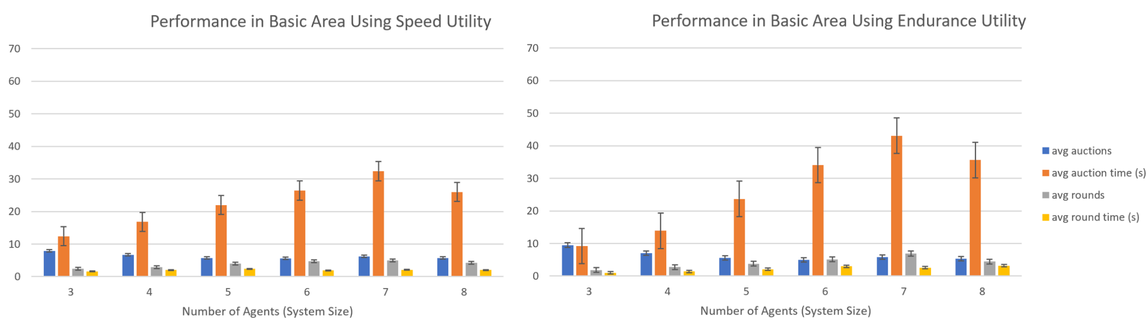

Figure 4.17. **Auction Performance in Basic Area.** This figure shows the auction performance in our basic area for the speed utility (left) and endurance utility (right) functions. The greater the number of agents participating in the auctions, the longer the auctions take.

As system size grows, average auction runtimes grow as well. This is caused by at least two things. First, increased competition for limited resources causes the system to require more rounds to complete assignment. As more agents bid for fewer cells, the competition generates bidding wars that, by definition, create a single winning bidder per round and thus require more rounds to achieve unique cell-agent pairs. Having more auction rounds leads to longer auction times even though round times remain low regardless of system size.

The second reason auction runtimes increase as system size increases is the increased burden placed on the synchronization framework introduced in Section 3.3. As more agents are cooperating in the system, more messages are transmitted and the risk of message collisions and data loss increases. This increased messaging and re-transmission burden increases auction times because the system can only proceed as quickly as the last agent receiving a required update. Figure 4.18 shows our implementation's runtime and worktime performance across system sizes as compared to the perfect search.

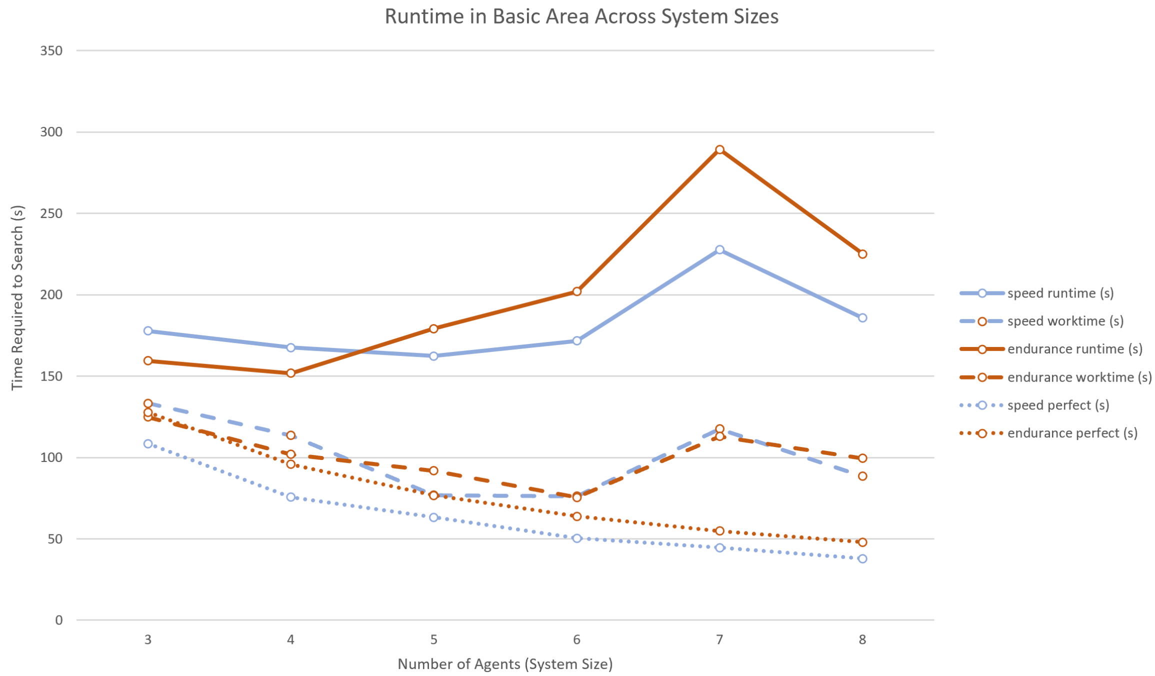

Figure 4.18. **AuctionSearch Runtimes and Worktimes in Basic Area.** This figure shows how much time is required to search the basic area. Increased auction times contribute to increased runtime in our implementation as system size increases because auctions are initiated at cell completion. Other trigger strategies that reduce time spent loitering will achieve improved search times.

While our implementation requires more time than the perfect search, worktime generally decreases with an increase in system size. This point is more apparent in the larger environments, specifically the large area, where less competition for cells occurs. Transit time contributes to longer runtimes as well in the basic area at system sizes seven and eight. Our implementation seeks to maximize per-agent utilization through its utility functions' deference to less capable agents (i.e., slow or low-endurance agents) when utility costs are low, as in the basic area. Therefore less capable agents are allowed to transit further to maximize their per-robot utility in the basic area, contributing to the increased runtime and worktime. This result is less pronounced in the complex and large areas due to the larger utility cost variances encountered.

In Chapter 5 we provide more detail regarding ways to mitigate auction impact on overall search runtimes, such as starting auctions based on different trigger criteria than cell completion. Increased messaging has no impact on the total number of auctions, however, which decreases as system size increases. This occurs because there are more search cells assigned per auction, reducing the total number of auctions.

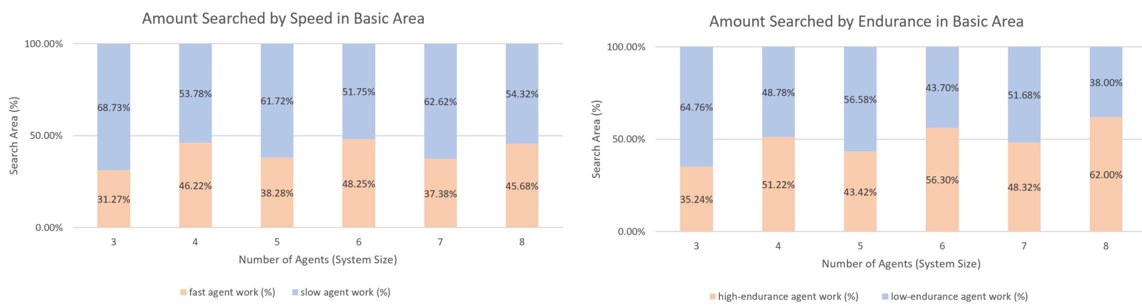

Figure 4.19. **`AuctionSearch` Division of Work in Basic Area.** This figure shows the percentage searched by highly capable versus less capable agents across system sizes. As discussed in 4.4, we assigned even-numbered agents high speed and endurance and odd-numbered agents were assigned low speed and endurance in order to achieve a roughly 50 percent split per system size.

Figure 4.19 shows the percentages of work completed in the basic area. Our speed utility function, shown on the left side of Figure 4.19, shows the expected result of fast agents affording slow agents the opportunity to take lower-cost cells. The basic area is so small, however, that fast agents defer most of the search to slow agents and, therefore, never have the opportunity to take larger-cost cells; A behavior which is shown to be achieved in our complex and large areas.

Our endurance utility function results, shown on the right side of Figure 4.19, show a more equitable division of labor in the basic area. This occurs because low-endurance agents are only presented with low cost cells due to the small area to be search, and therefore were not penalized as harshly as in the other, larger search areas.

4.5.4 `AuctionSearch` Performance Against the Perfect Search

Measuring `AuctionSearch` against variants of itself, albeit a valuable tuning strategy, provides no basis from which to measure its performance against other auction-based implementations. Comparing performance to the perfect search, however, provides a common benchmark for all auction-based area search applications to measure against.

The perfect search makes many assumptions and represents ideal circumstances which rarely, if ever, materialize in real implementation. System dynamics such as communication infrastructure, decentralized assignment deliberation, and orphan cell management impact performance in less than ideal, real-world applications. Regardless, certain design decisions make auction-based implementations perform more closely to the ideal than others.

Figure 4.20 shows how `AuctionSearch`'s worktimes compare to the perfect search across all configurations.

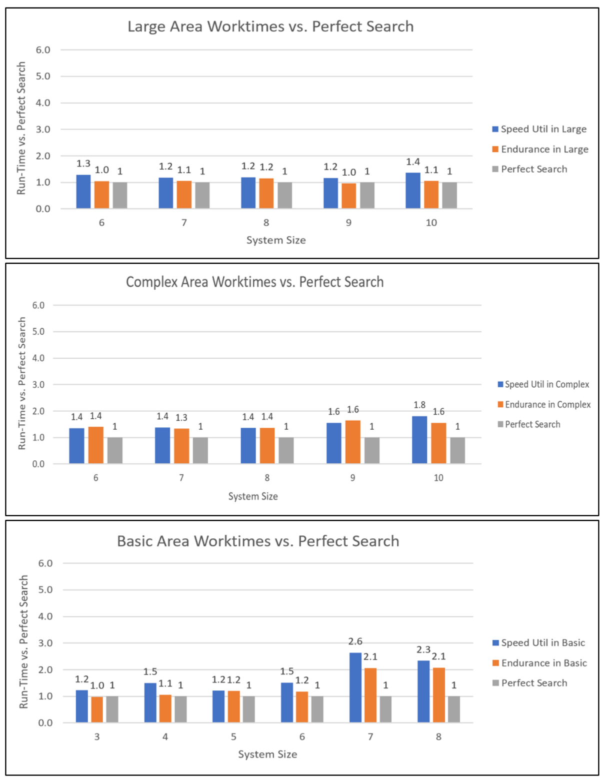

Figure 4.20. **`AuctionSearch` Worktimes Versus the Perfect Search.** This figure shows how many times longer our implementation took to completely search across all configurations than the theoretical perfect search. Our implementation fared best in the large area where competition for cells is lowest. It fared worst in the basic area where competition is highest, especially at larger system sizes.

Our implementation performed closest to the perfect search in the large area where auction durations were low due to minimal bidding conflicts among the many cells. Performance suffered the most in the basic area where the agents quickly outnumber the cells, creating many cell conflicts that result in increased auction duration averages.

4.5.5 Live-Fly Results

In this Subsection, we discuss and compare the results of live `AuctionSearch` experimentation conducted at Camp Roberts, CA with the NPS ARSENL Zephyr II UAVs. We ran `AuctionSearch` on two separate occasions with different system sizes and utility functions in the basic area to validate our simulation results and verify operation with real-world constraints. We were only able to conduct live experimentation with the basic area due to range safety restrictions.

We conducted live-flight demonstration in August 2018 with system sizes ranging from three to six agents to demonstrate our implementation's real-world feasibility. We then conducted live-flight testing in November 2018 with system sizes ranging from four to eight agents and both utility functions to validate and compare the results against our simulation results. Figure 4.21 shows the auction statistics for live-flight across system sizes for both utility functions.

The trend of increased system sizes having increased average auction runtimes, overall runtimes, and worktimes is evident in live-flight as it was in simulation. Of note, we were only able to test each system size one time, so the results presented here do not benefit from results averaged over time. It is assumed that live-flight results would track closer to simulated results given more runs, as the standard deviation in values is within the same range observed in simulation.

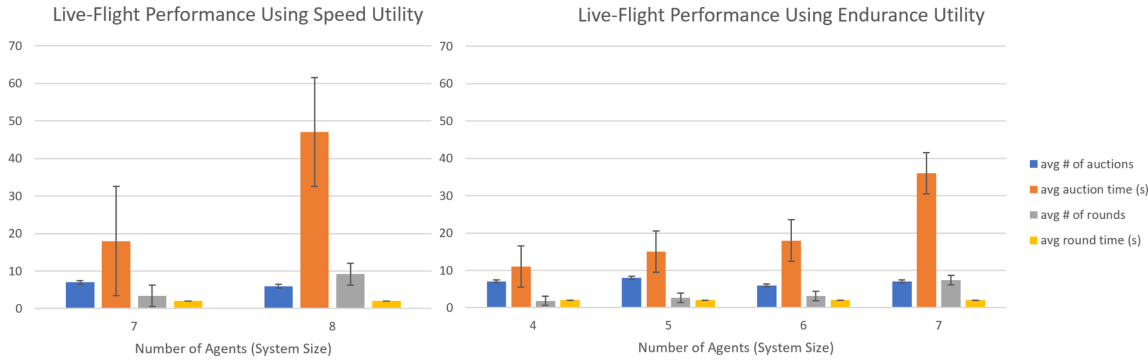

Figure 4.21. **Live-Flight Performance in Basic Area.** This figure shows how our implementation performs in live experimentation conducted at Camp Roberts, CA. The average auction runtimes increase as system-size increases, as was seen in simulation.

Figure 4.22 shows the live-flight run times of `AuctionSearch` in the basic area across system sizes for both utility functions. These runtimes are graphed with the perfect search for comparison. Our live-flight results closely match the trend evident in simulation, showing that average runtimes increase with system-size for our implementation.

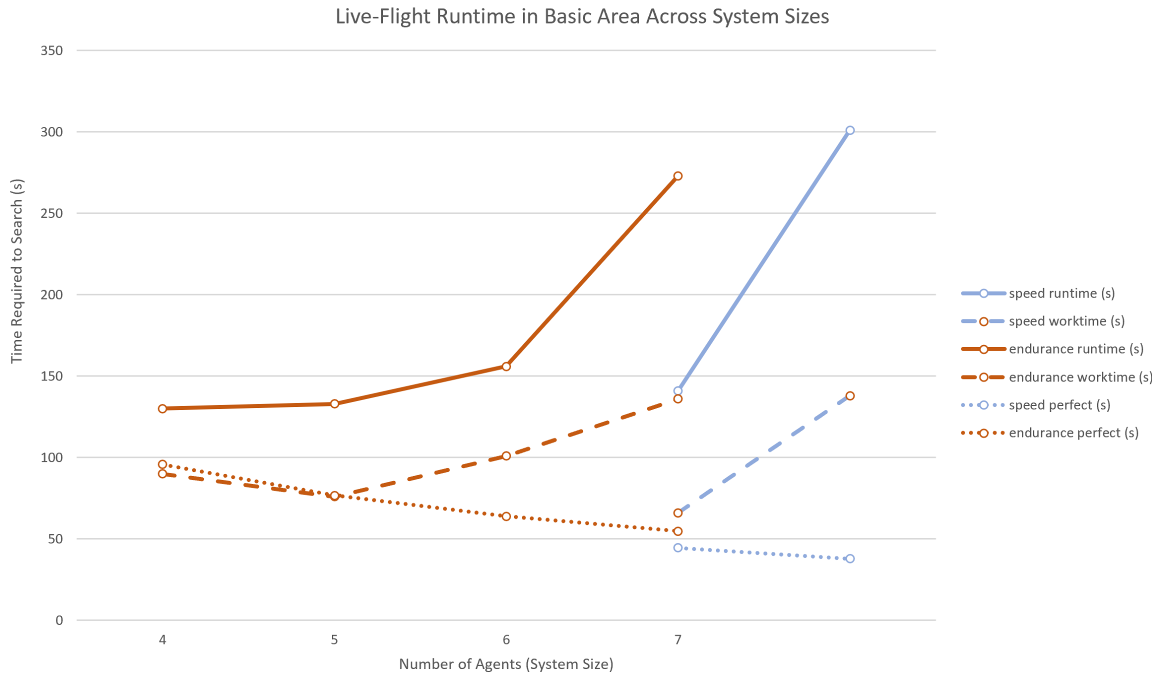

Figure 4.22. **AuctionSearch Live-Flight Runtimes and Worktimes in Basic Area.** This figure shows how much time is required to search the basic area in our live-flight conducted at Camp Roberts, CA. Increased auction times, as shown in Figure 4.22, contribute to increased runtime as system size increases in our implementation.

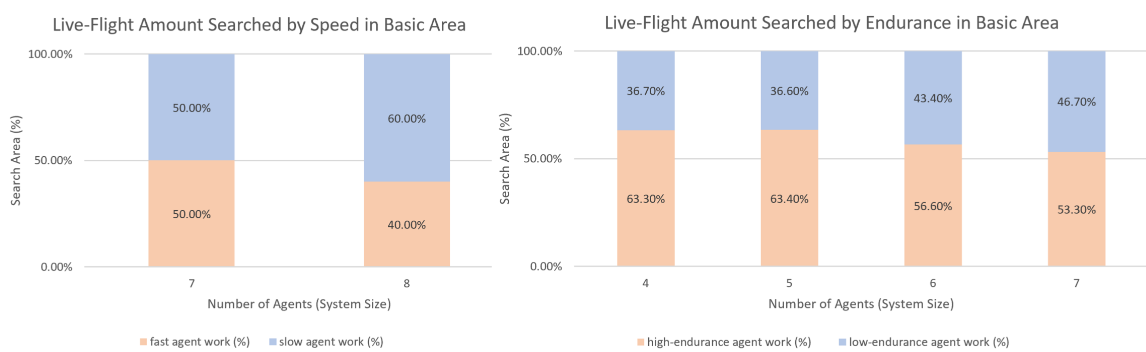

Figure 4.23. **Live-Flight AuctionSearch Division of Work in Basic Area.** This figure shows the percentage searched by fast or high-endurance versus slow or low-endurance agents across system sizes during live-flight experimentation at Camp Roberts, CA. Highly capable agents conducted the majority of the search, but a more or less equitable split was achieved as intended.

Figure 4.23 shows the percentage split of work completed by fast or slow agents (using the speed utility function) and by high and low-endurance agents (using the endurance utility function) in the basic area across system sizes. Again, live-flight experimentation yields similar results to simulation.

4.6 Summary

In this chapter we introduced our speed and endurance utility functions that seek to achieve agent bidding strategies that maximize system-wide utilization and efficiency in auction-based area search. We also discussed the impacts that these functions have on agent bids and the expected behavior from each. Our utility functions allow less capable agents (i.e., slow or low-endurance agents) the opportunity to make maximum use of their available time and energy while more capable agents (i.e., fast or high-endurance agents) take on larger and harder work that their less capable counterparts cannot efficiently complete.

We defined the measures of performance for our implementation and conducted 326 simulated and live runs in three different areas with system sizes ranging from three to 10 robots each. The results of these various experiment configurations provide a basis for using auction-based cell assignment in area search applications in distributed systems. We observed how different cellular decomposition strategies achieved different results with various system sizes and mixes of high and low-capability agents. We also observed how the burdens of system-wide data consistency and distributed autonomy impact overall runtime of auction-based implementations. We then validated our simulated results with live-flight testing of our implementation.

With all data collected and results analyzed, we present our findings, conclusions, and recommendations in Chapter 5. We also discuss the lessons learned during design, development, and testing of `AuctionSearch` and outline what future research would inform the area of autonomous decision in multi-robot systems.

CHAPTER 5:
Conclusion

The overarching goal of this thesis was to demonstrate the applicability of auction-based assignment for the autonomous execution of area search by multi-robot systems. This research goal was achieved by first exploring the different variations of market-based assignment algorithms and their application to area search. We then introduced `AuctionSearch`; our single-item auction-based area search behavior implemented for the ARSENL multi-UAV system.

We went on to discuss the design trade-offs required to implement auction-based area search and subject `AuctionSearch` to a wide range of tests, spanning three search areas, two utility functions, and system sizes varying from three to 10 robots each. Per-robot and system-wide statistics were collected and analyzed to measure `AuctionSearch`'s performance as an auction-based area search solution. We compared our results for various configurations to identify those aspects having the largest impact on search performance. Finally, we validated our simulation-environment results with live-fly field experimentation to assess performance with regard to the challenges presented by interaction with hardware the real world.

5.1 Findings and Lessons Learned

In this section, we discuss our findings regarding auction-based area search and provide recommendations for optimization given our implementation and research objectives. Our `AuctionSearch` implementation described in Chapter 3 and its experimentation and test results presented in Chapter 4 show that auction algorithms are well suited for autonomous area search applications with multi-robot systems. Our research shows that satisfactory cell assignment solutions can be achieved with auction algorithms. Our research also shows that multi-robot systems are capable of achieving complete area search autonomously using auction algorithms, and can do so in restrictive, lossy-communications environments of various size and complexity.

5.1.1 Increasing the Number of Agents Generally Decreases Search Times

Worktime, which we defined in Chapter 4 as our implementation's overall runtime with the loiter time removed, generally decreases when more agents are included in an area search. While auction-based assignment takes a non-trivial amount of time to achieve optimal cell-agent pairs, it can perform nearly as well as the perfect search with optimal auction-start criteria and agent bidding (or "no-bidding") strategies.

To achieve area search runtimes that are close to that of the perfect search, agents need to be searching cells for as much of the area search runtime as possible. This requires that agents always have assigned cells, implying that auctions for those assignments occur and are concluded prior to the completion of the current *in_progress* cell. This can be achieved by requiring agents to start the next auction immediately after setting a cell to *in_progress* or upon reaching some *remaining_length* threshold for their current cell instead of waiting until a cell is set to *complete*, as our implementation does. Search runtimes might also be decreased by assigning more of the search area per auction using combinatorial auctions.

Intra-system communication frequency and message complexity can also contribute to increased runtimes. Communication frequency directly impacts runtime based on the amount of built-in redundancy. For systems such as ours that are required to operate in lossy-communications environments, the allowances to account for lost data are mandatory, to prevent data inconsistency from undermining system-wide consensus. Messaging schemes such as ours that use requests as the fail-safe for missed messages will incur some lost time while retransmissions occur. This equates to some level of inefficiency, but it makes the system robust to data-loss and ensures consensus is maintained throughout execution. We view this trade-off as acceptable and have made the intentional decision to prioritize correctness over efficiency.

5.1.2 Increasing the Number of Agents Increases Auction Duration

Auctions take longer to achieve unique cell-agent pairs as the number of participating agents increases. While our research shows that having more agents generally decreases area search completion times, particularly in large areas with many search cells, individual auction performance suffers as a result of having more agents. Increased auction times are a natural byproduct of having more agents vying for the same number of cells, particularly as the number of cells continues to decrease toward the end of a search when the agents begin to outnumber the cells. Our research indicates that there may be an optimal number of agents for a given search area which, if surpassed, degrades search performance. We leave the actual identification of the optimal number given per-robot capabilities and cellular decomposition, however, to future work.

Efforts can be undertaken to lessen the impact of increased competition, such as providing off-ramps for subsets of agents who fall below some utility threshold or are bidding back and forth between two cells of similar utility. This will allow highly competitive auctions to complete more quickly but may result in more sub-optimal cell-agent pairs. Our experiments indicate that some degree of sub-optimality may be preferable to extended auctions, however, since it allows agents to commence work more quickly. Further, increased optimality can once again be pursued in subsequent auctions that reshuffle any *assigned* or *available* cells.

5.1.3 Having Many Small Cells Increases Efficiency in `AuctionSearch`

The results of our large-area experiments indicate that our single-item auction assignment scheme and utility functions lend themselves best to having many small cells as opposed to fewer larger cells. When there are large numbers of small cells, agents have many fallback options if they do not bid successfully for their favorite cell in a given round. This leads to shorter auction times since agents settle on cell assignments more quickly.

The opposite effect was observed in the basic- and complex-area experiments. Agents have very few options among the 12 and 16 cells, respectively, and therefore spend large amounts of time in auction competing for the small number of cells. The prevalence of extremely large cells in the complex area results in steep utility dropoff rates which cause agents to pursue their favorite cell repeatedly to avoid being forced to accept their second favorite assignment. This extended pursuit equates to increased auction times and sometimes even inefficient assignment solutions. As more agents are introduced, the competition for the few small cells is exacerbated, and auction times increase.

5.1.4 Utility Function Modification can Achieve a Range of Bidding Behaviors

Our results demonstrate that using agent characteristics to calculate cell utility can modify agent bidding strategies by influencing what each agent views as a high-utility task. Modification of agent bidding strategies at the individual level appears to be the most tuneable aspect of auction-based search implementation. A wide range of strategies can be implemented to achieve efficient results across a variety of system sizes in as wide a range of possible search areas.

For our research we used speed and endurance to achieve equitable division of labor across all search areas to maximize per-robot utilization. Our implementation combines all aspects of utility cost (*distance*, *size*, and *remaining_size*) and uses agent speed or endurance as the differentiator between agents' utility values for a particular cell. More nuanced variable control might be used to refine bidding strategies further. For example, increases in per-robot utilization can likely be achieved by incentivizing agents based on their particular *distance* value. That is, agents can be made to favor closer cells regardless of size, producing a greedy system response that may be desirable for certain applications. Other agent characteristics that can be considered for more nuanced auction-based assignment include sensor, defensive, and offensive capabilities.

5.2 Future Work

This thesis explored the technical capability of multi-robot systems to conduct area search operations without human intervention. Our work implemented auction-based assignment to achieve this level of autonomy, but it hardly represents the limit to which the field of autonomous decision and distributed robotics should explore. Future research efforts should experiment with a more broad spectrum of algorithms, system configurations, and mission sets to identify more avenues for maturation of autonomous decision approaches. Further, future efforts should explore how deeply these algorithms can be nested and linked to develop ever-more robust behaviors.

5.2.1 Autonomous Decision Using Combinatorial Auctions

Combinatorial auctions are as well-suited as single-item auctions to area search problems, if not more so. Future research should explore the efficiency achieved by conducting auction-based assignment of search cells using combinatorial auctions with various numbers of cells per awardable subset. Many of the same design trade-offs wrestled with in `AuctionSearch` will need to be addressed with combinatorial auctions as well, and many more issues will require attention as well given the subset selection difficulties discussed in Chapter 2.

5.2.2 Independent Cuing and Nesting of Auctions During Area Search

One paradigm that should be explored is the linking of autonomous decision frameworks, like auction algorithms, to allow multi-robot systems to perform arbitrarily complex behaviors given operational triggers, specific and specialized agent capabilities, and knowledge of desired end-states. Area searches, for instance, are rarely ordered as stand alone operations. Rather, they are typically information-gathering endeavors that inform follow-on actions. If the agents participating in the area search know what operational outcomes are desired and have knowledge of their own capabilities, the distributed system can collectively determine how best to execute any number of trigger-based follow-on tasks.

Auctions can be initiated via agent-to-agent cuing with the goal of assigning some number of agents to a single objective requiring attention. While some subset of agents bids for and executes the emergent task (e.g., attack, follow, report, or defend), the remaining agents can detect this through the auction process and dynamically reassign area search cells among themselves. This would push even more autonomy to the per-robot level, testing the discovering agent's ability to not only identify, classify, and individually execute an objective, but to also alert any number of the other agents to come to its aid on that objective. In this way, arbitrary linkages of tasks can be achieved to develop arbitrarily complex behaviors in a robust and failure-tolerant fashion.

Work in this area should not be completed in a vacuum or without an overarching framework to govern systems' employment for arbitrarily complex missions. The Mission-Based Architecture for Swarm Composability (MASC) introduced by [51] represents a mission-focused systems engineering framework within which technical implementation and experimentation can be undertaken to avoid haphazard and unfocused autonomous systems behavior development. Ultimately, the more autonomy that can be pushed to the edge of our multi-robot systems, the more capable those systems will be for undertaking complex behaviors.

APPENDIX: **AuctionSearch** Source Code

```python
'''
----------------------------------------------------------------------------

- AuctionSearch
- Matthew S. Hopchak, 2018

- Area search behavior utilizing an auction algorithm to autonomously distribute
  search cells among the swarm participants.

- This file contains four classes to run auction-based area search
    1. AuctionSearch:  conduct auctions for cell assignments at certain intervals
       and run an area search given a number of agents and a search area.
    2. Cell: hold, mutate, and access attributes of cell objects for conduct of area search and auctions.
    3. Waypoint:  hold, mutate, and access attributes of waypoint objects for conduct of area search.]
    4. Searcher: hold, mutate, and access attributes of searcher objects for conduct of area search,
       participation in auctions, and communication with other agents.

----------------------------------------------------------------------------
'''
from __future__ import division
```

```
import math
import rospy
import ap_msgs.msg as apmsg
import std_msgs.msg as stdmsg
import ap_lib.gps_utils as gps
import ap_lib.math_utils as ro_math
import ap_lib.nodeable as nodeable
import ap_lib.ap_enumerations as enums
import ap_lib.bitmapped_bytes as bytes
import ap_lib.plugin_behavior as plugin
import ap_mission_planning.swarm_manager as swarm

class AuctionSearch(plugin.PluginBehavior):
    ''' Area search swarm behavior using auction algorithms
    Used to distribute search cells from a given search area to particpant
    agents and to autonomously assign new cells at certain trigger intervals.
    Auctions are initiated each time an agent completes a cell to maximize
    system-wide utilization.  Agents use either the speed or the endurance
    utility functions to compute utility values for each cell, and subsequently
    bid for their favorite (highest utility-yielding) cell.  If they are outbid,
    agents pursue other cells or the same cell again, increasing their bid, depending
    on utility.  The algorithm terminates once there are no more cells left requiring
    search.

    Member variables:
```

```
        _agent: Searcher instantiation for holding searcher information like cells owned
        _search_roll_call: set to keep track of agents who have reported cell statuses
        _bid_roll_call: set to keep track of agents who have reported bids
        _been_there: set to keep track of cells that have been searched (are completed)
        _cells_left: set to keep track of cells left to be searched (are available or in-progress)
        _round_tracker: set used by syncRounds() to track what round all agents are in for consistency
        _cells_in_progress: set to keep track of cells that are in-progress
        _cells_not_won: set used to keep track of cells an agent has lost this auction
        _cells_changed: set used to keep track of cells that have updated statuses
        _cell_update_sent: set to track which cells an agent has sent updates for
        _abandoned_cells: set used to keep track of which cells an agent has stopped pursuing this auction
        _complete_roll_call: set used to keep track of agents who have reported auction complete
        _loiter_checkpoint: list to hold latitude and longitude of last waypoint for loiter location
        _inbound_statuses: list of lists of cell statuses received from other agents before processing
        _west_wall: list holding cartesian coordinates to western boundary of the search area
        _north_wall: list holding cartesian coordinates to northern boundary of the search area
        _east_wall: list holding cartesian coordinates to
```

```
    eastern boundary of the search area
_south_wall: list holding cartesian coordinates to
    southern boundary of the search area
_obstacle_grids: list holding cartesian coordinates
    to the vertices of each obstacle in complex area
_obstacles: list of lists holding vertex-edge-vertex
     for each obstacle in complex area
_inbound_bids: list of tuples of bids received from
    other agents
_cell_utilities: list of tuples of cell utilities
    within a given round of an auction
_curr_bid: list containing the cell_id and bid_value
     for a given cell in a current auction round
_prev_bid: list containing the cell_id and bid_value
     of my previous bid to allow agents to catch up
_prev_cells: list of cell_ids that changed last
    auction round which require communication of
    updates
_all_bids: dictionary of bids for a round where key
    == searcher_id and value == (cell_id:bid_value)
_cells: dictionary to hold all cell objects
_message_count: counter of number of cell status
    requests the agent has sent this auction
_bid_msg_count: counter of number of bid status
    requests the agent has sent this round
_auc_msg_count: counter of number of auction
    complete requests the agent has sent this auction
_sync_msg_count: counter of number of sync requests
    the agent has sent this round
_auction_number: counter of number of auctions
_round_number: counter of number of rounds for a
    particular auction
_wait: counter used to meter how often underutilized
```

```
            agents request auction
        _cell_memory: counter used to meter how many
            previous rounds of cell changes agents maintain
        _sensor_sweep: list containing the waypoint spread
            by step (m) and stride (m)
        _search_area: Geobox object containing the southwest
             lat/long, width, and height of search area
        _rounds_synced: boolean flag for whether agents can
            proceed with round
        _loiter_wait: boolean flag for whether agent must
            stay at current waypoint
        _bids_updated: boolean flag for whether agent has
            received updated bids from all agents
        _initial_assign: boolean flag for whether it is the
            very first auction
        _winners_picked: boolean flag for whether agent has
            completed winner determination
        _mid_search_bid: boolean flag for whether agent has
            submitted a bid, completing obligation
        _submit_same_bid: boolean flag for whether agent won
             its last round and needs to submit same bid
        _same_bids: boolean flag for whether all agents
            submitted the same bids
        _auction_started: boolean flag for whether agent is
            in an auction
        _bidding_complete: boolean flag for whether agent
            has submitted a bid
        _auction_complete: boolean flag for whether agent
            has completed an auction
        _cell_complete: boolean flag for whether agent has
            completed an in-progress cell
        _i_finished_last: boolean flag for whether agent
            finished the last cell of the search, to tell
```

```
    others
_agent._IS_SEARCHER: boolean flag for whether
    behavior is active with cells left to search
_cell_update_complete: boolean flag for whether
    agent has received updated cell statuses from
    others
_agent._IS_SEARCH_AUCTION: boolean flag for whether
    agent is in an auction
_choose_search_area: AuctionSearch enumeration for
    basic, large-basic, or complex search area
_choose_utility_function: AuctionSearch enumeration
    for speed or endurance utility function
_data_auction_durations: data capture: no bearing on
    AuctionSearch operation
_data_round_durations: data capture: no bearing on
    AuctionSearch operation
_data_round_information: data capture: no bearing on
    AuctionSearch operation
_data_robot_searching: data capture: no bearing on
    AuctionSearch operation
_data_robot_loitering: data capture: no bearing on
    AuctionSearch operation
_data_robot_utilization: data capture: no bearing on
    AuctionSearch operation
_data_total_runtime: data capture: no bearing on
    AuctionSearch operation
_data_round_time: data capture: no bearing on
    AuctionSearch operation
_data_auction_time: data capture: no bearing on
    AuctionSearch operation
_data_area_searched: data capture: no bearing on
    AuctionSearch operation
_total_search_waypoints: data capture: no bearing on
```

```
                AuctionSearch operation
            _am_searching: data capture: no bearing on AuctionSearch operation
            _am_loitering: data capture: no bearing on AuctionSearch operation

        Inherited member variables (PluginBehavior):
            id: Unique integer identifier for this behavior
            manager: BehaviorManager object to which this behavior belongs

        Member functions:
            parameterize: implementation of the Behavior virtual function
            compute_command: runs one iteration of the behavior's control loop
            safety_checks: completes behavior-specific safety checks
            process_behavior_data: process various behavior messages
            auctionCompleteRequest: execute requests for auction completion (lossy comms protection)
            auctionStatusRequest: execute requests for auction status (lossy comms protection)
            bidStatusUpdate: send bids to other agents
            bidStatusRequest: execute requests for bids (lossy comms protection)
            calculateUtility: calculate the utility for a given cell for a given agent
            calculateUtilityCost: calculate distance, size, and remaining_size for a given cell
            calcTotalArea: used for data collection. No bearing on AuctionSearch execution
```

```
captureRobotUtilizationData: used for data collection. No bearing on AuctionSearch execution
captureRoundData: used for data collection. No bearing on AuctionSearch execution
captureThesisData: used for data collection. No bearing on AuctionSearch execution
cellStatusUpdate: send cell status updates to other agents
cellStatusRequest: execute requests for cell statuses (lossy comms protection)
checkIfAuctionComplete: check if auction is complete and tell other agents if so
checkIfCellComplete: check if agent completed an in-progress cell, and start an auction for new cells
checkIfSearchComplete: check if the search is complete, and start an auction to notify others if so
checkUtilization: check if there are cells available even though agent has none. Start an auction.
consolidateBids: place bids from inbound_bids list into a dictionary for processing
defineGeometries: define the edges of obstacles in the complex area
determineOffLimits: determine which cells are not available for auciton
determineWaypoint: determine which waypoint to travel to, or loiter at
displayReport: display auction and assignment information
displayShortReport: display round and bid information
externalUpdateMyCells: update knowledge of cells
```

```
    from other agents' knowledge
finalAuction: starts one more auction if agent was
    last to finish search
finishAuction: clean up data structures after an
    auction has finished
fromWaypoint: determine grid to use as last waypoint
     for utility cost calculations
generateAdjacencyGraph: creates neighbors lists for
    each cell
generateBasicCells: creates cell objects of
    rectangular shape of specified height/width (m)
generateBasicSearchArea: fills boundary data
    structures given basic, large, or complex area
generateCellAssignment: assign a cell won in auction
     to an agent
generateCellUtilities: calculate the utility an
    agent gains for owning a cell
generateComplexSearchCells: creates cell objects of
    polygonal shape given obstacle locations
generateSearchBid: calculate an agent's bid for a
    cell given utility calculations
generateWaypoints: create waypoint objects in a cell
     given sweep width (spread, stride) (m)
getInTheAuction: start an auction and reinitialize
    all associated data structures
internalUpdateCells: update local cell knowledge
    given winning bids from an auction
makeCellActive: set an assigned cell to in-progress
    once a waypoint has been reached
moveToNextCell: move to an assigned cell upon
    completion of in-progress cell
reassignCell: change assignment of a cell from one
    agent to another
```

```
        removeCellAssignment: change cell status to
            assignment-removed so other agents can detect it
        revertCell: change cell status from assignment-
            removed to available
        sendAuctionComplete: send a single message telling
            other agents that agent is finished with auction
        setWaypoint: send a speed waypoint command message
            with lat/lon/alt/speed information
        shareAuctionComplete: lossy-comms tolerant way to
            reliably communicate auction status with agents
        shareBids: lossy-comms tolerant way to reliably
            communicate bids with agents
        shareStatuses: lossy-comms tolerant way to reliably
            communicate cell statuses with agents
        startAuction: send a burst of auction start messages
             to other agents
        stayInMyCell: command agent to loiter at last
            waypoint after finishing its cell
        submitSearchBid: send a single message telling other
             agents bid information
        syncRounds: check whether all agents are in the same
             round, behind, or ahead in an auction
        testWaypoint: check whether an agent has arrived at
            a specified waypoint
        winnerDetermination: determine highest bidder from a
             set of bids and direct auction termination

    Inherited member functions (PluginBehavior)
        set_ready: safely sets the ready state to True or
            False
        is_ready: returns the behavior's current readiness
            state
    '''
```

```
    # 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

    # Utility Function enumerations
    SPEED_UTIL = 3
    ENDURANCE_UTIL = 4

    # 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
    CAPTURE_DIST   = 65
    MESSAGE_COUNT  = 20
    EPSILON        = 300
    NUM_CELLS      = 12   # this is modified by the cell generation methods below
    NOT_BIDDING    = NUM_CELLS
    CELLS_PER_AUCTION = 2
    CELL_STATUS_MEMORY= 4

    def __init__(self, behavior_id, behavior_name, manager=None):
```

```
        ''' Class initializer initializes class variables.
        @param behavior_id: unique identifier for this
            behavior
        @param behavior_name: string name of this behavior
        @param manager: BehaviorManager object to which this
             behavior belongs
        '''
        plugin.PluginBehavior.__init__(self, behavior_id,
            behavior_name, manager)
        self._agent = Searcher(rospy.get_param("aircraft_id"
            ))
        self._search_roll_call = set()
        self._bid_roll_call    = set()
        self._been_there       = set()
        self._cells_left       = set()
        self._round_tracker    = set()
        self._cells_in_progress= set()
        self._cells_not_won    = set()
        self._cells_changed    = set()
        self._cell_update_sent = set()
        self._abandoned_cells  = set()
        self._complete_roll_call = set()
        self._loiter_checkpoint= [ ]
        self._inbound_statuses = [ ]
        self._west_wall      = [ ]
        self._north_wall     = [ ]
        self._east_wall      = [ ]
        self._south_wall     = [ ]
        self._obstacle_grids= [ ]
        self._obstacles      = [ ]
        self._inbound_bids   = [ ]
        self._cell_utilities= [ ]
        self._curr_bid       = [ ]
```

```
        self._prev_bid       = [ ]
        self._prev_cells     = [ ]
        self._all_bids       = { }
        self._cells          = { }
        self._message_count = 0
        self._bid_msg_count = 0
        self._auc_msg_count = 0
        self._sync_msg_count= 0
        self._auction_number= 0
        self._round_number  = 0
        self._wait          = 0
        self._cell_memory   = 0
        self._sensor_sweep  = [75, 75]
        self._search_area   = None
        self._rounds_synced = True
        self._loiter_wait   = False
        self._bids_updated  = False
        self._initial_assign= True
        self._winners_picked = False
        self._mid_search_bid    = False
        self._submit_same_bid   = False
        self._same_bids         = False
        self._auction_started   = False
        self._bidding_complete = False
        self._auction_complete = False
        self._cell_complete     = False
        self._i_finished_last   = False
        self._agent._IS_SEARCHER = True
        self._cell_update_complete = False
        self._agent._IS_SEARCH_AUCTION = True
        self._choose_search_area = AuctionSearch.
            BASIC_LIVE_FLY
        self._choose_utility_function = AuctionSearch.
```

```
SPEED_UTIL
        # data capture instrumentation follows:  no bearing on AuctionSearch operation
        self._data_auction_durations = [ ]
        self._data_round_durations   = [ ]
        self._data_round_information = [ ]
        self._data_robot_searching   = [ ]
        self._data_robot_loitering   = [ ]
        self._data_robot_utilization = [ ]
        self._data_total_runtime     = [ ]
        self._data_round_time        = [ ]
        self._data_auction_time      = [ ]
        self._data_area_searched     = 0.0
        self._total_search_waypoints = 0
        self._am_searching           = False
        self._am_loitering           = False

    #-----------------------------------------------
    # Implementation of parent class virtual functions
    #-----------------------------------------------

    def parameterize(self, params):
        ''' Sets behavior parameters based on set service parameters and speed/endurance values
        Parameters for AuctionSearch include:
        _choose_search_area: enumeration identifying which search area it being used
        _choose_utility_function: enumeration identifying which utility function agents should use
        @param params: parameters from the set service request
        @return True if set with valid parameters
        '''
```

```
        self.manager.log_info("initializing auction searcher")

        # reinitialize all __init__ parameters for subsequent run
        self._search_roll_call.clear()
        self._bid_roll_call.clear()
        self._been_there.clear()
        self._cells_left.clear()
        self._round_tracker.clear()
        self._cells_in_progress.clear()
        self._cells_not_won.clear()
        self._cells_changed.clear()
        self._cell_update_sent.clear()
        self._abandoned_cells.clear()
        self._complete_roll_call.clear()
        self._loiter_checkpoint= []
        self._inbound_statuses = []
        self._west_wall      = []
        self._north_wall     = []
        self._east_wall      = []
        self._south_wall     = []
        self._obstacle_grids= []
        self._obstacles      = []
        self._inbound_bids   = []
        self._cell_utilities= []
        self._curr_bid       = []
        self._prev_bid       = []
        self._prev_cells     = []
        self._all_bids       = {}
        self._cells          = {}
        self._message_count = 0
        self._bid_msg_count = 0
```

```
        self._auc_msg_count = 0
        self._sync_msg_count= 0
        self._auction_number= 0
        self._round_number  = 0
        self._wait          = 0
        self._cell_memory   = 0
        self._sensor_sweep  = [75, 75]    # [waypoint spread, ceiling/floor distances]
        self._search_area   = None
        self._rounds_synced = True
        self._loiter_wait   = False
        self._bids_updated  = False
        self._initial_assign= True
        self._winners_picked = False
        self._mid_search_bid   = False
        self._submit_same_bid  = False
        self._same_bids        = False
        self._auction_started  = False
        self._bidding_complete = False
        self._auction_complete = False
        self._cell_complete    = False
        self._i_finished_last  = False
        self._agent._IS_SEARCHER = True
        self._agent._IS_SEARCH_AUCTION = True
        self._cell_update_complete = False
        self._agent.resetCurrWaypointId()
        self._agent.removeAllAssignments()
        self._choose_search_area = AuctionSearch.BASIC_LIVE_FLY
        self._choose_utility_function = AuctionSearch.SPEED_UTIL
        # data capture instrumentation follows: no bearing on AuctionSearch operation
```

```
        self._data_auction_durations = [ ]
        self._data_round_durations   = [ ]
        self._data_round_information = [ ]
        self._data_robot_searching   = [ ]
        self._data_robot_loitering   = [ ]
        self._data_robot_utilization = [ ]
        self._data_total_runtime     = [ ]
        self._data_round_time        = [ ]
        self._data_auction_time      = [ ]
        self._data_area_searched     = 0.0
        self._total_search_waypoints = 0
        self._am_searching           = False
        self._am_loitering           = False


        # ---------- EXPERIMENT VARIABLES. MODIFY THESE ---------------------


        #     1. Choose Search Area. -------------------------------------
        #     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
        # ----- SELECT ONE OF THE BELOW OPTIONS ----
        self._choose_search_area = AuctionSearch.BASIC_LIVE_FLY
        #self._choose_search_area = AuctionSearch.BASIC_LARGER
        #self._choose_search_area = AuctionSearch.COMPLEX
```

```
        #     2.  Choose Utility Function. -----------------------------------
        #     SPEED_UTIL:      Generate private value (utility) using individual speeds
        #     ENDURANCE_UTIL:  Generate private value (utility) using individual endurance
        # ----- SELECT ONE OF THE BELOW OPTIONS ----
        self._choose_utility_function = AuctionSearch.SPEED_UTIL
        #self._choose_utility_function = AuctionSearch.ENDURANCE_UTIL


        # ---------- END EXPERIMENT VARIABLES. DO NOT MODIFY BELOW -----------

        self._data_total_runtime.append(rospy.Time.now())
        # generate the outer search area boundary for any chosen search area
        self.generateBasicSearchArea()

        # If search area is complex (includes obstacles), conduct boustrophedon cellular decomposition
        if self._choose_search_area >= AuctionSearch.COMPLEX:
            self.manager.log_info("Generating Complex Search Area Parameters")
            AuctionSearch.NUM_CELLS   = self.generateComplexSearchCells()
            AuctionSearch.NOT_BIDDING = AuctionSearch.NUM_CELLS
            self.generateAdjacencyGraph()
```

```python
            self.set_ready(True)
        elif self._choose_search_area in [AuctionSearch.BASIC_LIVE_FLY, AuctionSearch.BASIC_LARGER]:
            self.manager.log_info("Generating Basic Search Area Parameters")
            self.generateAdjacencyGraph()
            self.set_ready(True)
        else:
            self.manager.log_info("Unrecognized search area enumeration used.  Shutting down.")
            self.set_ready(False)

        # set total number of waypoints for thesis data capture
        self.calcTotalArea()

        # set individual utility variables based on user-selected utility function
        if self._choose_utility_function == AuctionSearch.SPEED_UTIL:
            self.manager.log_info("Speed Utility Function Chosen.  Setting Agent Speeds.")
            # All agents with an even searcher_id are faster than odds
            if self._agent.getSearcherId() % 2 == 0:
                self._agent.setSpeed(23)
            else:
                self._agent.setSpeed(15)
            self.set_ready(True)
        elif self._choose_utility_function == AuctionSearch.ENDURANCE_UTIL:
            self.manager.log_info("Endurance Utility Function Chosen.  Setting Agents' Endurance."
```

```
                )
            # All agents with an even searcher_id have
                higher endurance than odds
            if self._agent.getSearcherId() % 2 == 0:
                self._agent.setEndurance(0.8)
            else:
                self._agent.setEndurance(0.2)
            self._agent.setSpeed(15)
            self.set_ready(True)
        else:
            self.manager.log_info("Unrecognized␣utility␣
                function␣enumeration␣used.␣␣Shutting␣down.")
            self.set_ready(False)

        # if the number of agents is greater than half the
            number of cells,
        #  change how many cells per auction to expect (
            pigeon hole)
        if AuctionSearch.NUM_CELLS / 2 < len(self.manager.
            subswarm_keys):
            AuctionSearch.CELLS_PER_AUCTION = 1

        # initialize first waypoint
        self._loiter_checkpoint = [self.manager.
            get_own_state().state.pose.pose.position.lat,
                                    self.manager.
                                        get_own_state().state.
                                        pose.pose.position.lon
                                        ]
        self._data_auction_time.append(rospy.Time.now())

        return self.is_ready()

```

```python


    def process_behavior_data(self, data_msg):
        ''' receive and direct action based on data messages received from other agents
        Parsers for these data messages are contained in bitmapped_bytes.py
        '''
        if data_msg.id == bytes.AUCTION_BID:
            # I have received a message containing a bid for a cell
            parsed = bytes.AuctionSearchBidParser()
            parsed.unpack(data_msg.params)
            self._round_tracker.add(parsed.round_id)
            if parsed.round_id == self._round_number or parsed.bid_value == AuctionSearch.NOT_BIDDING:
                if parsed.source_id not in self._bid_roll_call and not self._bids_updated and \
                  self._bidding_complete:
                    if parsed.bid_cell_id == AuctionSearch.NOT_BIDDING:
                        self.manager.log_info("agent %d says hes not bidding." % parsed.source_id)
                    bid_val = int(round(parsed.bid_value))
                    self._inbound_bids.append( [parsed.source_id, parsed.bid_cell_id, bid_val] )
                    self._bid_roll_call.add(parsed.source_id)
            if self._i_finished_last:
```

```
                self._i_finished_last = False

        elif data_msg.id == bytes.AUCTION_BIDS_REQUEST:
            # see if the request is for the previous round or not to allow an agent to catch up
            parsed = bytes.UShortParser()
            parsed.unpack(data_msg.params)
            if parsed.value >= self._round_number:
                is_previous = False
            else:
                is_previous = True
            for i in range(2):
                self.bidStatusUpdate(is_previous)

        elif data_msg.id == bytes.AUCTION_STATUS:
            parsed = bytes.AuctionStatusParser()
            parsed.unpack(data_msg.params)
            if parsed.auction_number >= self._auction_number:
                self._round_tracker.add(parsed.round_number)

        elif data_msg.id == bytes.AUCTION_NEW:
            parsed = bytes.NewAuctionParser()
            parsed.unpack(data_msg.params)
            if not self._auction_started and self._auction_complete and \
               self._auction_number == parsed.auction_number:
                self.getInTheAuction()
                # if agent who started the auction claims next cell, set it to IN_PROGRESS locally
                if parsed.claim_next_cell:
                    self._cells[parsed.next_cell_id].setStatus(Cell.IN_PROGRESS)
```

```
                    # thesis data capture line
                    if len(self._data_robot_searching) == 0:
                        self._data_robot_searching.append(rospy.Time.now())
                        self._am_searching = True
                    self._cells_changed.add(parsed.next_cell_id)
                    self._cells_in_progress.add(parsed.next_cell_id)

        elif data_msg.id == bytes.AUCTION_CELLS:
            parsed = bytes.AuctionSearchCellsParser()
            parsed.unpack(data_msg.params)
            if parsed.source_id not in self._search_roll_call:
                self._inbound_statuses.append( [parsed.source_id, parsed.cell_list] )
                self._search_roll_call.add(parsed.source_id)

        elif data_msg.id == bytes.AUCTION_CELLS_REQUEST:
            parsed = bytes.AuctionStatusParser()
            parsed.unpack(data_msg.params)
            auction_num = parsed.auction_number
            round_num   = parsed.round_number
            # If I missed the auction start message, set auction start (redundency for lossy comms)
            if parsed.round_number == 0 and not self._auction_started:
                self.manager.log_info("missed the auction start message.  Catching up.")
                self.manager.log_info("sending requested cell status.")
                self.cellStatusUpdate()
```

```
                self.getInTheAuction()
            else:
                self.cellStatusUpdate()

        elif data_msg.id == bytes.AUCTION_COMPLETE:
            parsed = bytes.UShortParser()
            parsed.unpack(data_msg.params)
            agent_id = parsed.value
            if agent_id not in self._complete_roll_call:
                self._complete_roll_call.add(agent_id)
                self.manager.log_info("Agent %d reports auction complete" % parsed.value)

        elif data_msg.id == bytes.AUCTION_COMPLETE_REQUEST:
            parsed = bytes.UShortParser()
            parsed.unpack(data_msg.params)
            am_complete = False
            if len(self._agent.getMyCellIds()) == AuctionSearch.CELLS_PER_AUCTION and \
              self._initial_assign:
                am_complete = True
            elif self._mid_search_bid:
                am_complete = True
            elif not self._initial_assign and not self._mid_search_bid and \
              parsed.value < self._auction_number:
                am_complete = True
            if am_complete:
                for i in range(AuctionSearch.MESSAGE_COUNT):
                    self.sendAuctionComplete()
```

```
    def compute_command(self):
        ''' Executes one iteration of the behavior
        Agents start off requiring an auction for cells. Once cells are assigned, agents can
        execute auctions and search at the same time. Behavior is finished once all cells have
        been searched
        '''
        # capture thesis data
        self.captureRobotUtilizationData()

        num_agents = len(self.manager.subswarm_keys)
        # auction for search cells state functionality below #
        if self._agent._IS_SEARCH_AUCTION:
            self._rounds_synced = self.syncRounds(num_agents)
            if self._rounds_synced:
                if not self._cell_update_complete:
                    self.shareStatuses(num_agents)
                elif self._cell_update_complete and not self._mid_search_bid:
                    if not self._auction_complete:
                        if not self._bidding_complete:
                            self.generateSearchBid()
                        if self._bidding_complete:
                            if not self._bids_updated:
                                self.shareBids(num_agents)
                            elif self._bids_updated:
                                self.winnerDetermination()
                if not self._auction_complete:
                    self.checkIfAuctionComplete(num_agents)
                if self._auction_complete and self.
```

```
_cell_update_complete:
                    self.finishAuction()
                    self.displayReport()
            elif not self._rounds_synced:
                self.manager.log_info("I am ahead and need to wait. agent %d, round is %d. " \
                                      % (self._agent.getSearcherId(), self._round_number))

        # area searcher state functionality below #
        if self._agent._IS_SEARCHER:
            self.checkIfCellComplete()
            self.checkIfSearchComplete()
            waypoint_data = self.determineWaypoint()
            self.setWaypoint(waypoint_data[1])
            self.testWaypoint(waypoint_data[1])
            self.checkUtilization()

        # if the search is complete, the last agent starts a final auction to ensure
        #   all agents terminate gracefully
        self.finalAuction()
        return self.manager.spd_wp_cmd_msg



    def safety_checks(self):
        ''' Conducts behavior-specific safety checks
        @return True if the behavior passes all safety checks (False otherwise)
        '''
```

```python
        # if agent is in one of these four states, return True.
        if self._agent._IS_SEARCHER or self._agent._IS_SEARCH_AUCTION or \
          len(self._been_there) == AuctionSearch.NUM_CELLS:
            return True
        # or as long as the agent is still in the sub-swarm, return True
        elif self._agent.getSearcherId() in self.manager.subswarm_keys:
            return True
        else:
            self.manager.log_warn("agent failed AuctionSearch safety checks.")
            return False
        return True



    #-----------------------------------------------
    # Behavior-specific methods in alphabetical order
    #-----------------------------------------------


    def auctionCompleteRequest(self):
        ''' execute requests for auction completion (lossy comms protection)
        '''
        parser = bytes.UShortParser()
        parser.value = self._auction_number
        report = self.manager.behavior_data_msg
        report.id = bytes.AUCTION_COMPLETE_REQUEST
        report.params = parser.pack()
```

```
            self.manager.behavior_data_publisher.publish(report)



        def auctionStatusRequest(self):
            ''' execute requests for auction status (lossy comms
                protection)
            '''
            parser = bytes.AuctionStatusParser()
            parser.auction_number = self._auction_number
            parser.round_number   = self._round_number
            report = self.manager.behavior_data_msg
            report.id = bytes.AUCTION_STATUS
            report.params = parser.pack()
            self.manager.behavior_data_publisher.publish(report)



        def bidStatusUpdate(self, is_previous):
            ''' send bids to other agents during an auction
            @param is_previous: boolean flag for whether the
               requesting agent is a round behind
            '''
            parser = bytes.AuctionSearchBidParser()
            parser.source_id = self._agent.getSearcherId()
            # if an agent has fallen behind and is trying to
               catch up, send previous bid
            if is_previous:
                bid = self._prev_bid
                round_id = self._round_number - 1
            else:
                bid = self._curr_bid
                round_id = self._round_number
```

```
        if len(bid) > 0:
            parser.round_id    = round_id
            parser.bid_cell_id = bid[0]
            parser.bid_value   = int(round(bid[1]))
            report = self.manager.behavior_data_msg
            report.id = bytes.AUCTION_BID
            report.params = parser.pack()
            self.manager.behavior_data_publisher.publish(
                report)



    def bidStatusRequest(self):
        ''' execute requests for bids (lossy comms
            protection)
        '''
        parser = bytes.UShortParser()
        parser.value = self._round_number
        report = self.manager.behavior_data_msg
        report.id = bytes.AUCTION_BIDS_REQUEST
        report.params = parser.pack()
        self.manager.behavior_data_publisher.publish(report)



    def calculateUtility(self, cell_id, last_waypoint):
        ''' calculate the utility for a given cell for a
            given agent
        @param cell_id: id of the cell
        @param last_waypoint: the waypoint to start distance
             calculations from
        @return the utility for the specified cell from the
            specified waypoint
```

```
        '''
        utility_cost  = self.calculateUtilityCost(cell_id, last_waypoint)
        if cell_id == self._agent.getCurrCellId():
            cell_cost = 0.0
        else:
            cell_cost = self._cells[cell_id].getCost()
        private_value = self._cells[cell_id].getValue()
        if self._choose_utility_function == AuctionSearch.SPEED_UTIL:
            util_cost_time = utility_cost / self._agent.getSpeed()
            cell_utility   = round(private_value - util_cost_time - cell_cost, 3)
        elif self._choose_utility_function == AuctionSearch.ENDURANCE_UTIL:
            util_cost_endur = utility_cost / (10 * self._agent.getEndurance())
            cell_utility    = round(private_value - util_cost_endur - cell_cost, 3)
        return cell_utility




    def calculateUtilityCost(self, cell_id, last_waypoint):
        ''' used by generateCellUtilities() to generate all relevant components
            for the utility calculation
        @param cell_id: id of the cell
        @param last_waypoint: the waypoint to start distance calculations from
        @return the utility cost for specified cell
```

```
        '''
        self._cells[cell_id].deleteWaypoints()
        self.generateWaypoints(cell_id, last_waypoint)
        cell_location = self._cells[cell_id].getWaypoints()[0].getLatLonLocation()
        cell_size = self._cells[cell_id].getSize()
        if len(self._agent.getMyCellIds()) > 0:
            if self._cells[self._agent.getCurrCellId()].getStatus() == Cell.ASSIGNED:
                curr_cell_left = len(self._cells[self._agent.getCurrCellId()].getWaypoints()) \
                                 * self._sensor_sweep[1]
            else:
                curr_cell_left = (len(self._cells[self._agent.getCurrCellId()].getWaypoints()) \
                                  - self._agent.getCurrWaypointId()) * self._sensor_sweep[1]
            dist_to_cell = gps.gps_distance(cell_location[0], cell_location[1], \
                                            last_waypoint[0], last_waypoint[1])
        else:
            curr_cell_left = 0
            dist_to_cell   = gps.gps_distance(cell_location[0], cell_location[1], \
                                              last_waypoint[0], last_waypoint[1])
        # sum terms to produce the gross utility (before
```

```
subtracting cell cost)
        utility_cost = dist_to_cell + cell_size + curr_cell_left
        return utility_cost



    def calcTotalArea(self):
        ''' for data collection.  determine total number of waypoints in any cell.
        '''
        bot = self.manager.get_own_state().state.pose.pose.position
        start_position = (bot.lat, bot.lon)
        for cell_id in range(len(self._cells)):
            self.generateWaypoints(cell_id, start_position)
            self._total_search_waypoints += len(self._cells[cell_id].getWaypoints())
            self._cells[cell_id].deleteWaypoints()



    def captureRobotUtilizationData(self):
        ''' for data collection.  determine loiter, transit, and search times for each agent.
        '''
        if self._rounds_synced:
            if not self._initial_assign:
                # if i'm not searching, I'm either transiting or I'm loitering
                if not self._am_searching:
                    # if loiter_wait is true, I'm loitering
                    if self._loiter_wait and not self.
```

```
            _am_loitering:
                self._data_robot_loitering.append(rospy.Time.now())
                # set am_loitering to true to make sure we only append once
                self._am_loitering = True
            # if loiter_wait is false, I'm no longer loitering
            elif not self._loiter_wait and self._agent.getCurrCellId() != None and \
              self._am_loitering:
                self._data_robot_loitering.append(rospy.Time.now())
                # set am_loitering to false to make sure we only append once
                self._am_loitering = False
            # if I don't have a current cell, I am loitering
            elif self._agent.getCurrCellId() == None and len(self._cells_left) > 0 and \
              not self._am_loitering:
                self._data_robot_loitering.append(rospy.Time.now())
                # set am_loitering to true to make sure we only append once
                self._am_loitering = True
            # if search is over and I was still loitering, append second loiter value
            if len(self._cells_left) == 0 and not self._agent._IS_SEARCH_AUCTION and \
              self._am_loitering:
                self._data_robot_loitering.append(rospy.Time.now())
```

```
            self._am_loitering = False
    # if I've started searching, I'm not loitering or transiting
    elif self._am_searching:
        # if am_loitering is still true, need to append second loiter time
        if self._am_loitering:
            self._data_robot_loitering.append(rospy.Time.now())
            # set am_loitering to false to make sure we only append once
            self._am_loitering = False
elif self._initial_assign:
    # if we have just started the behavior, append the first loiter value
    if not self._am_loitering:
        self._data_robot_loitering.append(rospy.Time.now())
        self._am_loitering = True
# append durations when they become available
if len(self._data_robot_loitering) == 2:
    start_time = self._data_robot_loitering[0]
    end_time   = self._data_robot_loitering[1]
    total_loiter_time = end_time - start_time
    self._data_robot_utilization.append(("l", total_loiter_time))
    self._data_robot_loitering = [ ]
if len(self._data_robot_searching) == 2:
    start_time = self._data_robot_searching[0]
    end_time   = self._data_robot_searching[1]
    total_search_time = end_time - start_time
    self._data_robot_utilization.append(("s", total_search_time))
```

```
            self._data_robot_searching = [ ]



    def captureRoundData(self):
        ''' for data collection. capture start and end times of rounds.
        '''
        self._data_round_time.append(rospy.Time.now())
        if len(self._data_round_time) > 1:
            start_time = self._data_round_time[0]
            end_time   = self._data_round_time[1]
            round_time = end_time - start_time
            self._data_round_durations.append(round_time)
            self._data_round_time = [ ]
        if self._auction_complete:
            # calculate number of rounds and average round runtimes
            num_rounds = len(self._data_round_durations)
            # iterate through rospy time duration instances
            round_times = rospy.Duration(0)
            for duration in self._data_round_durations:
                round_times += duration
            if num_rounds > 0:
                average_round_time = round_times / num_rounds
            else:
                average_round_time = rospy.Duration(0)
            self._data_round_information.append((num_rounds, average_round_time))
            self._data_round_time = [ ]
            self._data_round_durations = [ ]

```

```
    def captureThesisData(self):
        ''' for data collection.  compile and display iteration data for collection
        '''
        # calculate total AuctionSearch runtime
        start_time = self._data_total_runtime[0]
        end_time   = self._data_total_runtime[1]
        total_runtime = end_time - start_time
        # calculate number of auctions and average auction runtimes
        num_auctions = len(self._data_auction_durations)
        # iterate through rospy time duration instances
        auction_times = rospy.Duration(0)
        for duration in self._data_auction_durations:
            auction_times += duration
        average_auction_time = auction_times / num_auctions
        # calculate rounds per auction and other round-count information
        total_rounds = 0
        total_round_times = rospy.Duration(0)
        for info in self._data_round_information:
            total_rounds += info[0]
            total_round_times += info[1]
        average_rounds = total_rounds / len(self._data_round_information)
        average_round_times = total_round_times / len(self._data_round_information)
        # calculate per-robot utilization
        total_loiter  = rospy.Duration(0)
        total_search  = rospy.Duration(0)
        total_transit = rospy.Duration(0)
```

```
        for capture in self._data_robot_utilization:
            if capture[0] == "s":
                total_search += capture[1]
            else:
                total_loiter += capture[1]
        total_transit = total_runtime - total_search - total_loiter
        percent_i_searched = self._data_area_searched / self._total_search_waypoints
        self.manager.log_info("Searcher_id = %d" % self._agent.getSearcherId())
        self.manager.log_info(("number of auctions: ", num_auctions))
        self.manager.log_info(("average auction duration: ", average_auction_time.secs))
        self.manager.log_info(("average rounds per auction: ", round(average_rounds, 2)))
        self.manager.log_info(("average round time per auction: ", average_round_times.secs))
        self.manager.log_info(("total runtime: ", total_runtime.secs))
        self.manager.log_info("")
        self.manager.log_info(("total time spent loitering: ", total_loiter.secs))
        self.manager.log_info(("total time spent transiting: ", total_transit.secs))
        self.manager.log_info(("percentage of area I searched: ", round(percent_i_searched, 3)))



    def cellStatusUpdate(self):
        ''' send cell status updates to other agents
```

```
        '''
        parser = bytes.AuctionSearchCellsParser()
        parser.cell_list = [ ]
        parser.source_id = self._agent.getSearcherId()
        parser.round_id  = self._round_number
        parser.auction_number = self._auction_number
        # send current cell updates as well as the most recent previous updates
        cells_to_transmit = self._cells_changed.union(self._prev_cells)
        if len(cells_to_transmit) != 0:
            for cell_id in cells_to_transmit:
                self._cell_update_sent.add(cell_id)
                parser.cell_list.append( [self._cells[cell_id].getCellId(), \
                                          self._cells[cell_id].getStatus(), \
                                          self._cells[cell_id].getOwner(),  \
                                          self._cells[cell_id].getCost()] )
        report = self.manager.behavior_data_msg
        report.id = bytes.AUCTION_CELLS
        report.params = parser.pack()
        self.manager.behavior_data_publisher.publish(report)



    def cellStatusRequest(self):
        ''' execute requests for cell statuses (lossy comms
```

```
            protection)
        '''
        parser = bytes.AuctionStatusParser()
        parser.auction_number = self._auction_number
        parser.round_number   = self._round_number
        report = self.manager.behavior_data_msg
        report.id = bytes.AUCTION_CELLS_REQUEST
        report.params = parser.pack()
        self.manager.behavior_data_publisher.publish(report)



    def checkIfAuctionComplete(self, num_agents):
        ''' check if auction is complete and tell other
            agents if so
        @param num_agents: the number of agents in the
            subswarm executing AuctionSearch
        '''
        if self._winners_picked:
            if len(self._agent.getMyCellIds()) ==
                AuctionSearch.CELLS_PER_AUCTION and \
               self._initial_assign:
                self.shareAuctionComplete(num_agents)
            elif self._mid_search_bid:
                self.shareAuctionComplete(num_agents)



    def checkIfCellComplete(self):
        ''' check if agent completed an in-progress cell,
            and start an auction for new cells
        '''
        if len(self._agent.getMyCellIds()) > 0:
```

```
cell_index = self._agent.getCurrCellId()
# if I have completed searching my cell, move to next cell or wait until I get a new cell
if self._agent.getCurrWaypointId() > 0 and len(self._cells[cell_index].getWaypoints()) > 0 \
  and self._cells[cell_index].getStatus() == Cell.IN_PROGRESS:
    if self._agent.getCurrWaypointId() > len(self._cells[cell_index].getWaypoints())-1 \
      and not self._cell_complete and not self._loiter_wait:
        self._cells[cell_index].setStatus(Cell.COMPLETE)
        # thesis data capture
        if len(self._data_robot_searching) == 1:
            self._data_robot_searching.append(rospy.Time.now())
            self._am_searching = False
        self._cells_changed.add(cell_index)
        self._loiter_checkpoint = self._cells[cell_index].getWaypoints()[-1].getLatLonLocation()
        self._cell_complete = True
        self._agent.resetCurrWaypointId()
        self.manager.log_warn("I have completed search of cell %d" % cell_index)
        # if I have finished the last cell in the search area, remember it to start auction
        if cell_index in self._cells_left and len(self._cells_left) in [0, 1]:
            self._i_finished_last = True
```

```
                    # capture thesis data
                    self._data_area_searched += len(self._cells[cell_index].getWaypoints())
                    if len(self._agent.getMyCellIds()) <= 1:
                        self.stayInMyCell()



    def checkIfSearchComplete(self):
        ''' check if the search is complete, and start an auction to notify others if so
        '''
        if len(self._cells_left) == 0 and not self._initial_assign and \
          not self._auction_started and self._agent._IS_SEARCHER:
            self.manager.log_info("Search is complete.  Deactivate Behavior.")
            self._cells_in_progress.clear()
            # start a final auction to force a cell status update informing all agents of completion
            more_to_search = False
            self.startAuction(more_to_search)
            self._agent.resetCurrWaypointId()
            self._agent._IS_SEARCHER = False



    def checkUtilization(self):
        ''' check if there are cells available even though agent has none.  Start an auction.
        '''
        if not self._auction_started and not self.
```

```
            _initial_assign and \
            len(self._agent.getMyCellIds()) == 0:
            # if there are cells left to search that are not in progress, start an auction
            if len(self._cells_left) > len(self._cells_in_progress) and \
                self._wait >= (AuctionSearch.MESSAGE_COUNT * 20):
                self.manager.log_info("Cells are available and I have none. Starting auction.")
                more_to_search = False
                self.startAuction(more_to_search)
            else:
                self._wait += 1  # self._wait gives other agents a chance to finishAuction()



    def consolidateBids(self):
        ''' place bids from inbound_bids list into a dictionary for processing
        '''
        cells_bid_on  = { }    # member-test cell_ids to check for conflicts
        # consolidate other agents' bids in _all_bids dictionary
        for update in self._inbound_bids:
            agent_key = update[0]
            cell_key  = update[1]
            bid_val   = update[2]
            # check for cell conflicts (two or more agents bidding for same cell)
            if cell_key in cells_bid_on:
```

```
                self._same_bids = False
            else:
                if cell_key != AuctionSearch.NOT_BIDDING:
                    cells_bid_on[cell_key] = 0 # we only care about fast lookup of cell_id
            # if first time seeing agent's bid or its for a new cell, create agent:{cell_id:bid} pair
            if agent_key not in self._all_bids or cell_key not in self._all_bids[agent_key]:
                self._same_bids = False # agent submitted a new bid
                self._all_bids[agent_key] = { }
                self._all_bids[agent_key][cell_key] = bid_val
            # if agent has bid higher for same cell, it is still not happy.
            elif bid_val != self._all_bids[agent_key][cell_key]:
                self._same_bids = False # agent submitted a new bid
                self._all_bids[agent_key][cell_key] = bid_val
            else:
                self._all_bids[agent_key][cell_key] = bid_val
        # include agent's bid into _all_bids, following same logic as other agents' bids
        agent_key = self._agent.getSearcherId()
        cell_key  = self._curr_bid[0]
        bid_val   = self._curr_bid[1]
        # check for cell conflicts with my bid included
        if cell_key in cells_bid_on:
            self._same_bids = False
```

```
        else:
            if cell_key != AuctionSearch.NOT_BIDDING:
                cells_bid_on[cell_key] = 0 # we only care about fast lookup of cell_id
        if agent_key not in self._all_bids or cell_key not in self._all_bids[agent_key]:
            self._same_bids = False
            self._all_bids[agent_key] = { }
            self._all_bids[agent_key][cell_key] = bid_val
        elif bid_val != self._all_bids[agent_key][cell_key]:
            self._same_bids = False
            self._all_bids[agent_key][cell_key] = bid_val
        else:
            self._all_bids[agent_key][cell_key] = bid_val
        # conduct one more sanity check for cell conflicts
        ids = [ ]
        cell_conflict = False
        for agent in self._all_bids:
            for cell in self._all_bids[agent]:
                if cell not in ids:
                    if cell != AuctionSearch.NOT_BIDDING:
                        ids.append(cell)
                else:
                    cell_conflict = True
                    break
        if cell_conflict:
            self._same_bids = False


    def defineGeometries(self, objects):
        '''
        Defines edges of geometries in COMPLEX environments (obstacles, no_fly_zones, etc.)
```

```
        @param objects: list of obstacle grids
        @return list of node-node connections for each
           obstacle
        '''
        geometries = [ ]
        each_geometry = [ ]
        for geo in objects:
            i = 0
            each_geometry = [ ]
            while i < len(geo)-1:
                each_geometry.append( (geo[i], geo[i+1]) )
                i += 1
            each_geometry.append( (geo[0], geo[-1]) )
            geometries.append(each_geometry)
        return geometries



    def determineOffLimits(self):
        '''
        used by self.generateCellUtilities() to decide which
            cells should be off limits
          during a given auction.  in-progress and complete
             cells are not auctionable
        @return list of cell_ids that are not available for
           auction
        '''
        if self._initial_assign:
            off_limits = (Cell.IN_PROGRESS, Cell.COMPLETE,
               Cell.ASSIGNED)
        else:
            off_limits = (Cell.IN_PROGRESS, Cell.COMPLETE)
        return off_limits
```

```python
    def determineWaypoint(self):
        ''' determine which waypoint to travel to, or loiter
            at
        '''
        claim_my_next_cell = False
        # if search is complete, orbit in place
        if not self._agent._IS_SEARCHER:
            waypoint_loc = self._loiter_checkpoint
        elif self._initial_assign and self._agent.
            getCurrCellId() == None:
            waypoint_loc = self._loiter_checkpoint
        # if I completed my cell, decide to move to next
            cell or wait
        elif self._cell_complete:
            # if we're in the middle of an auction and I
                have finished a cell, stay put
            if self._auction_started:
                self.stayInMyCell()
                waypoint_loc = self._loiter_checkpoint
            else:
                # otherwise, if my next cell is still
                    optimal, move to it
                if len(self._agent.getMyCellIds()) > 1 \
                  and self._agent.getMyCellIds()[1] in self.
                      _cells[self._agent.getCurrCellId()].
                      getNeighbors():
                    self.moveToNextCell()
                    claim_my_next_cell = True
                    waypoint_loc = self._cells[self._agent.
                        getCurrCellId()].getWaypoints()[self.
```

```
                            _agent.getCurrWaypointId()].
                            getLatLonLocation()
                    else:
                        self.stayInMyCell()
                        waypoint_loc = self._loiter_checkpoint
                # start a new auction for cells, and broadcast whether I claim my next cell or not (if it is still optimal)
                if not self._auction_started:
                    self.manager.log_info(("starting an auction.  claim_next_cell: ", claim_my_next_cell))
                    self.startAuction(claim_my_next_cell)
                self._cell_complete = False
            # if I have not completed my current cell, stay on the path to my current waypoint
            else:
                if self._agent.getCurrCellId() != None and \
                  self._agent.getCurrWaypointId() > len(self._cells[self._agent.getCurrCellId()].getWaypoints()):
                    waypoint_loc = self._loiter_checkpoint
                elif self._agent.getCurrCellId() != None and \
                  len(self._cells[self._agent.getCurrCellId()].getWaypoints()) > 0:
                    waypoint_loc = self._cells[self._agent.getCurrCellId()].getWaypoints()[self._agent.getCurrWaypointId()].getLatLonLocation()
                # if I don't have any assigned cells, loiter until I have a cell or until the search is complete
                else:
```

```
                waypoint_loc = self._loiter_checkpoint
        # if I've started searching my current cell, set it to IN_PROGRESS
        if self._agent.getCurrWaypointId() > 0 and not self._auction_started:
            self.makeCellActive()
        self._same_bids = False
        return (self._agent.getCurrWaypointId(), waypoint_loc)



    def displayReport(self):
        ''' display auction and assignment information
        '''
        self.manager.log_info(" ")
        self.manager.log_info(" ")
        self.manager.log_info(" ------------------------ ")
        self.manager.log_info("### ### NEW UPDATE ### ###")
        self.manager.log_info("Searcher_id = %d" % self._agent.getSearcherId())
        self.manager.log_info("Auction number = %d" % self._auction_number)
        self.manager.log_info("Auction round number = %d" % self._round_number)
        if self._choose_search_area in [AuctionSearch.BASIC_LIVE_FLY, AuctionSearch.COMPLEX]:
            self.manager.log_info("Search Cell Statuses are below.")
            self.manager.log_info("Format of each Cell Status is: [cell_id, cell_status, cell_owner, cell_cost]")
            rep = [ ]
```

```
        for i in range(len(self._cells)):
            rep.append( [self._cells[i].getCellId(), self._cells[i].getStatus(), \
                         self._cells[i].getOwner(), round(self._cells[i].getCost())] )
        self.manager.log_info(rep)
    self.manager.log_info(("cells_left:␣", self._cells_left))
    self.manager.log_info(("been_there:␣", self._been_there))
    self.manager.log_info(("cells_in_progress", self._cells_in_progress))
    if self._agent.getCurrCellId() != None:
        self.manager.log_info(("waypoints:␣", len(self._cells[self._agent.getCurrCellId()].getWaypointIds())))
        self.manager.log_info(("curr_waypoint:␣", self._agent.getCurrWaypointId()))
    self.manager.log_info(("curr_cell:␣", self._agent.getCurrCellId()))
    self.manager.log_info(("my␣cells:␣", self._agent.getMyCellIds()))
    self.manager.log_info("␣#␣#␣#␣#␣#␣#␣#␣#␣#␣#␣#␣#␣#␣#␣")
    self.manager.log_info("␣------------------------␣")
    self.manager.log_info("␣")
    self.manager.log_info("␣")
    if len(self._cells_left) == 0 and not self._initial_assign:
        self._data_total_runtime.append(rospy.Time.now())
        self._agent.resetCurrWaypointId()
```

```
            self.manager.log_info("Search is complete.  Deactivate Behavior.")
            # capture last loiter time for any orbiting agents for thesis data
            self.captureRobotUtilizationData()
            self.manager.log_info(" ")
            self.manager.log_info(" ")
            self.captureThesisData()
            self.manager.log_info(" ")
            self.manager.log_info(" ")



    def displayShortReport(self):
        ''' display round and bid information
        '''
        # provide a quick report of agent's information during each round
        self.manager.log_info("Bid info inside of winnerDetermination() is below")
        self.manager.log_info("Auction number = %d" % self._auction_number)
        self.manager.log_info("Auction round number: %d" % self._round_number)
        self.manager.log_info(("my_curr_bid: ", self._curr_bid))
        self.manager.log_info(("my cells: ", self._agent.getMyCellIds()))
        self.manager.log_info(("all_bids_dict: ", self._all_bids))
        self.manager.log_info(("cells_changed: ", self._cells_changed))
        self.manager.log_info(("prev_cells: ", self.
```

```
_prev_cells))
        self.manager.log_info("")



    def externalUpdateMyCells(self):
        ''' update knowledge of cells from other agents' knowledge
        '''
        removed_assignments = [ ]
        for update in self._inbound_statuses:
            if len(update[1]) > 0:
                for i in range(len(update[1])):
                    cell_id = update[1][i][0]
                    cell_status = update[1][i][1]
                    cell_owner  = update[1][i][2]
                    cell_cost   = update[1][i][3]
                    # assume new info based on another agent's higher cell status
                    if cell_status > self._cells[cell_id].getStatus():
                        self._cells[cell_id].setStatus(cell_status)
                        self._cells[cell_id].setOwner(cell_owner)
                        self._cells[cell_id].setCost(cell_cost)
                        # if another agent says a cell is complete, set that cell to complete
                        if cell_status == Cell.COMPLETE:
                            self._been_there.add(cell_id)
                            self._cells_in_progress.discard(
```

```
                            cell_id)
                        self._cells_left.discard(cell_id)
                # if the agent owning this cell is not active anymore, remove their assignment
                if cell_status == Cell.IN_PROGRESS:
                    if cell_owner not in self.manager.subswarm_keys:
                        self._cells[cell_id].setOwner(Cell.NO_OWNER)
                        self._cells[cell_id].setCost(Cell.NO_COST)
                        self._cells_in_progress.discard(cell_id)
                        self._cells_left.add(cell_id)
                        removed_assignments.append(cell_id)
                    else:
                        self._cells_in_progress.add(cell_id)
                elif cell_status == Cell.ASSIGNMENT_REMOVED \
                  and cell_id not in removed_assignments:
                    removed_assignments.append(cell_id)
        # to make cells available for auction again, update statuses.
        for cell_id in removed_assignments:
            self.revertCell(cell_id)
        removed_assignments = [ ]
```

```
        self._inbound_statuses = [ ]



    def finalAuction(self):
        ''' starts one more auction if agent was last to finish search
        '''
        if self._i_finished_last and not self._auction_started:
            if len(self._cells_left) == 0:
                if self._wait >= AuctionSearch.MESSAGE_COUNT * 3:
                    self._wait = 0
                    more_to_search = False
                    self.startAuction(more_to_search)
                    #self._i_finished_last = False
                else:
                    self._wait += 1



    def finishAuction(self):
        ''' clean up data structures after an auction has finished
        '''
        self._initial_assign   = False
        self._auction_started  = False
        self._loiter_wait      = False
        self._agent._IS_SEARCH_AUCTION = False
        self._prev_cells = [e for e in self._cells_changed]
        temp = self._cells_changed.difference(self._cell_update_sent)
```

```
        # clear the record of cell changes after a few auctions
        if self._cell_memory >= AuctionSearch.CELL_STATUS_MEMORY:
            self._cells_changed.clear()
            self._cell_memory = 0
        else:
            self._cell_memory += 1
        for e in temp:
            self._cells_changed.add(e)
        self._cell_update_sent.clear()
        self._search_roll_call.clear()
        self._complete_roll_call.clear()
        self._bid_roll_call.clear()
        self._abandoned_cells.clear()
        self._cells_not_won.clear()
        self._all_bids = { }
        self._wait = 0
        self._auc_msg_count = 0
        self._auction_number += 1
        # data capture instrumentation below
        self._data_auction_time.append(rospy.Time.now())
        auction_start_time = self._data_auction_time[0]
        auction_end_time   = self._data_auction_time[1]
        auction_runtime    = auction_end_time - auction_start_time
        self._data_auction_durations.append(auction_runtime)
        self._data_auction_time = [ ]
        self.manager.log_info("the auction is complete and cells are updated")
```

```
    def fromWaypoint(self, first_cell):
        ''' determine grid to use as last waypoint for utility cost calculations
        @return the waypoint from which distance calculations are started
        '''
        if self._initial_assign:
            if len(self._agent.getMyCellIds()) > 0 and \
              len(self._cells[first_cell].getWaypoints()) > 0:
                last_waypoint = self._cells[first_cell].getWaypoints()[-1].getLatLonLocation()
            else:
                bot = self.manager.get_own_state().state.pose.pose.position
                last_waypoint = (bot.lat, bot.lon)
        elif len(self._agent.getMyCellIds()) >= 1 and \
          self._cells[first_cell].getStatus() == Cell.ASSIGNED:
            bot = self.manager.get_own_state().state.pose.pose.position
            last_waypoint = (bot.lat, bot.lon)
        elif len(self._agent.getMyCellIds()) >= 1 \
          and self._cells[first_cell].getStatus() == Cell.IN_PROGRESS \
          and len(self._cells[first_cell].getWaypoints()) > 0:
            last_waypoint = self._cells[first_cell].getWaypoints()[-1].getLatLonLocation()
        else:
            bot = self.manager.get_own_state().state.pose.pose.position
            last_waypoint = (bot.lat, bot.lon)
```

```
        return last_waypoint



    def generateAdjacencyGraph(self):
        ''' creates 8-way adjacent neighbors lists for each
            cell
        '''
        # Finds the 8-way adjacency relationships of cells
            given a list of cells
        for i in range(len(self._cells)):
            cell = self._cells[i]
            for j in range(len(self._cells)):
                other = self._cells[j]
                common_bounds = False
                if cell.getCellId() != other.getCellId():
                    for grid in cell.getBoundaryGrids():
                        if grid in other.getBoundaryGrids():
                            common_bounds = True
                            break
                # if cell and other-cell share 1 boundary
                    grid, they are 8-way adjacent
                if common_bounds == True:
                    cell.addNeighbor(other.getCellId())
                    common_bounds = False



    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
        # Make these modified values publicly available
        AuctionSearch.NUM_CELLS    = l_divisor * w_divisor
        AuctionSearch.NOT_BIDDING = AuctionSearch.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)
        # Generate Cell objects using the length_lines (j) and width_lines (k) guidelines
        cell_id, j, k = 0, 0, 0
        temp_cell = None
        while cell_id < AuctionSearch.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._cells[cell_id] = temp_cell
                self._cells[cell_id].setWestBound((sw, nw))
                self._cells[cell_id].setEastBound((ne, se))
                self._cells_left.add(temp_cell.getCellId())
                k -= 1
                j += 1
                cell_id += 1
            k += 1
            j  = 0



    def generateBasicSearchArea(self):
        ''' fills boundary data structures given basic, large, or complex area
        '''
        if self._choose_search_area == AuctionSearch.BASIC_LIVE_FLY:
            cell_length = 200
            cell_width  = 200
        elif self._choose_search_area == AuctionSearch.BASIC_LARGER:
            cell_length = 325
            cell_width  = 325
        if self._choose_search_area == AuctionSearch.BASIC_LIVE_FLY:
            AuctionSearch.AREA_SW_LAT = 35.721147
            AuctionSearch.AREA_SW_LON = -120.773008
            AREA_ORIENT = 25.183537917993224
            AREA_LENGTH = 575
            AREA_WIDTH  = 750
```

```
        elif self._choose_search_area == AuctionSearch.BASIC_LARGER:
            AuctionSearch.AREA_SW_LAT = 35.721147
            AuctionSearch.AREA_SW_LON = -120.773008
            AREA_ORIENT = 0
            AREA_LENGTH = 2300
            AREA_WIDTH  = 3000
        elif self._choose_search_area == AuctionSearch.COMPLEX:
            AuctionSearch.AREA_SW_LAT = 35.72102
            AuctionSearch.AREA_SW_LON = -120.79111
            AREA_ORIENT = 0
            AREA_LENGTH = 2300
            AREA_WIDTH  = 3000
        else:
            self.manager.log_info("self._choose_search_area value unrecognized.  Area not created.")
        area = bytes.AuctionSearchBasicAreaParser()
        area.latitude     = AuctionSearch.AREA_SW_LAT
        area.longitude    = AuctionSearch.AREA_SW_LON
        area.length       = AREA_LENGTH
        area.width        = AREA_WIDTH
        area.orientation  = AREA_ORIENT
        self._search_area = gps.GeoBox(area.latitude, area.longitude, area.length, area.width, area.orientation)
        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.manager.log_info("Outer search area boundary generated.")
```

```
        if self._choose_search_area in [AuctionSearch.BASIC_LIVE_FLY, AuctionSearch.BASIC_LARGER]:
            self.generateBasicCells(AREA_LENGTH, AREA_WIDTH, cell_length, cell_width)



    def generateCellAssignment(self):
        ''' assign a cell won in auction to an agent
        '''
        bot = self.manager.get_own_state().state.pose.pose.position
        cell_id   = self._curr_bid[0]
        cell_cost = self._curr_bid[1]
        if len(self._agent.getMyCellIds()) > 0:
            assigned_cell = self._agent.getMyCellIds()[-1]
        # if the cell I just won is already assigned to me, update its cost with my current value
        if cell_id in self._agent.getMyCellIds():
            self.manager.log_info("Cell_id matches a cell I own.  Setting new cost.")
            self._cells[assigned_cell].setCost(cell_cost)
            self._cells_changed.add(assigned_cell)
            if self._agent.getCurrCellId() == assigned_cell:
                self._agent.resetCurrWaypointId()
        # if the cell I just won is different than my assigned cell, reassign to it
        else:
            self.manager.log_info("New cell_id assigned.")
            if len(self._agent.getMyCellIds()) > 1 and not self._initial_assign \
              and self._cells[assigned_cell].getStatus() != Cell.IN_PROGRESS:
```

```
                self.manager.log_info("Removing previous assignment.")
                self._cells[assigned_cell].deleteWaypoints()
                self.removeCellAssignment(assigned_cell)
                self._cells_changed.add(assigned_cell)
                self._agent.removeCell(assigned_cell)
            if len(self._agent.getMyCellIds()) == 0:
                self._agent.resetCurrWaypointId()
            self._agent.addCell(cell_id)
            self._cells[cell_id].setStatus(Cell.ASSIGNED)
            self._cells[cell_id].setOwner(self._agent.getSearcherId())
            self._cells[cell_id].setCost(cell_cost)
            self._cells_changed.add(cell_id)
        if len(self._agent.getMyCellIds()) == 1 and self._cells[cell_id].getStatus() == Cell.ASSIGNED:
            self._agent.resetCurrWaypointId()
            self.manager.log_info("I've been assigned a new cell.  Moving to cell.")
            self._loiter_wait = False



    def generateCellUtilities(self):
        ''' calculate the utility an agent gains for owning a cell
        '''
        self._cell_utilities = [ ]
        first_cell    = self._agent.getCurrCellId()  # if no current cell, first_cell == None
        last_waypoint = self.fromWaypoint(first_cell)
        off_limits    = self.determineOffLimits()
        for cell_id in self._cells_left:
```

```
            if self._cells[cell_id].getStatus() not in off_limits and cell_id not in self._abandoned_cells:
                if self._cells[cell_id].getStatus() == Cell.ASSIGNMENT_REMOVED:
                    self.revertCell(cell_id)
                cell_utility = self.calculateUtility(cell_id, last_waypoint)
                self._cells[cell_id].setUtility(cell_utility)
                self._cell_utilities.append( (cell_utility, cell_id) )
        # sort utility values. Highest utility (most valuable) in tuple (value, cell_id) at index [-1]
        self._cell_utilities.sort()



    def generateComplexSearchCells(self):
        '''
        Calculates Boustraphedon Decomposition given self.obstacles list in the following steps:
        1. Finds the outer perimeter of the environment
        2. Finds critical points and sorts them on x-value
        3. Cells are manually generated then instantiated as objects of a Cell class
        @return the number of cells to be searched
        '''
        bous_vertices = { }
        bous_cells    = [ ]
        # obstacle lat_lon corner locations
        obstacle_locations = [ ( (35.73800, -120.78375), (35.73918, -120.78545), (35.74049, -120.78332),
```

```
                        (35.73938, -120.78160) ), \
                                  ( (35.73270, -120.78571),
                                    (35.73408, -120.78702),
                                    (35.73558, -120.77891),
                                    (35.73319, -120.78005) ),
                                    \
                                  ( (35.72678, -120.78152),
                                    (35.72800, -120.78146),
                                    (35.72796, -120.77364),
                                    (35.72554, -120.77332) ),
                                    \
                                  ( (35.73110, -120.77050),
                                    (35.73627, -120.76903),
                                    (35.73205, -120.76780) ),
                                    \
                                  ( (35.72184, -120.76102),
                                    (35.72695, -120.76488),
                                    (35.72878, -120.76422),
                                    (35.72345, -120.75987) ),
                                    \
                                ]
        # convert lat_lon obstacle corners to cartesian x_y
            corners
        each_obstacle = [ ]
        for obstacle in obstacle_locations:
            for lat_lon in obstacle:
                x_y = gps.cartesian_offset(AuctionSearch.
                    AREA_SW_LAT, AuctionSearch.AREA_SW_LON,
                    lat_lon[0], lat_lon[1])
                each_obstacle.append(x_y)
            self._obstacle_grids.append(each_obstacle)
            each_obstacle = [ ]
        self._obstacles = self.defineGeometries(self.
```

```
            _obstacle_grids)
        # find the left and right critical points on each
            obstacle for boustrophedon cell boundaries
        left  = [0, 0]
        right = [0, 0]
        for obstacle in self._obstacle_grids:
            left  = [self._east_wall[0][0], self._east_wall
                [0][1]]
            right = [self._west_wall[0][0], self._west_wall
                [0][1]]
            for corner in obstacle:
                if corner[1] <= left[1]:
                    left[1] = corner[1]
                    left[0] = corner[0]
                if corner[1] >= right[1]:
                    right[1] = corner[1]
                    right[0] = corner[0]
            # add the boustraphedon critical vertices to a
                dictionary
            bous_vertices[left[0]]  = (left[0], left[1])
            bous_vertices[right[0]] = (right[0], right[1])
        # get the boustraphedon vertices into sorted order
        verts = [ ]
        size = len(bous_vertices)
        i = 0
        while i < size:
            mini = min(bous_vertices.keys())
            verts.append(bous_vertices[mini])
            del bous_vertices[mini]
            i += 1
        # find the obstacle intersections given the search
            area and obstacles
        inter1 = ro_math.segment_intersect( (self.
```

```
_north_wall[0][0], verts[8][1]), \
                                              (self._south_wall[0][0], verts[8][1]), \
                                               self._obstacles[1][1][0], self._obstacles[1][1][1] )
        inter2 = ro_math.segment_intersect( (self._north_wall[0][0], verts[9][1]), \
                                              (self._south_wall[0][0], verts[9][1]), \
                                               self._obstacles[1][1][0], self._obstacles[1][1][1] )
        inter3 = ro_math.segment_intersect( (self._north_wall[0][0], verts[2][1]), \
                                              (self._south_wall[0][0], verts[2][1]), \
                                               self._obstacles[1][3][0], self._obstacles[1][3][1] )
        inter4 = ro_math.segment_intersect( (self.
```

```
_north_wall[0][0], verts[7][1]), \
                                        (self._south_wall[0][0], verts[7][1]), \
                                         self._obstacles[2][1][0], self._obstacles[2][1][1] )
        # create the COMPLEX cell boundaries given obstacle locations
        cell_0 = [self._west_wall, (self._north_wall[0], (self._north_wall[0][0], verts[6][1])), \
                  ((self._north_wall[0][0], verts[6][1]), (self._south_wall[0][0], verts[6][1])), \
                  (self._south_wall[0], (self._south_wall[0][0], verts[6][1]))]
        cell_1 = [(verts[6], (self._north_wall[0][0], verts[6][1])), \
                  ((self._north_wall[0][0], verts[6][1]), (self._north_wall[0][0], verts[8][1])), \
                  ((self._north_wall[0][0], verts[8][1]), inter1), (inter1, verts[6])]
        cell_2 = [(verts[8], (self._north_wall[0][0], verts[8][1])), \
                  ((self._north_wall[0][0], verts[8][1]), (self._north_wall[0][0], verts[9][1])), \
                  ((self._north_wall[0][0], verts[9][1]), verts[9]), self._obstacles[0][2], self.
```

```
    _obstacles[0][1]]
        cell_3 = [(inter1, verts[8]), self._obstacles[0][0], self._obstacles[0][3], \
                (verts[9], inter2), (inter2, inter1)]
        cell_4 = [((self._south_wall[0][0], verts[6][1]), verts[6]), \
                self._obstacles[1][0], (self._obstacles[1][0][0], inter3), \
                (inter3, (self._south_wall[0][0], verts[2][1])), \
                ((self._south_wall[0][0], verts[2][1]), (self._south_wall[0][0], verts[6][1]))]
        cell_5 = [((self._south_wall[0][0], verts[2][1]), verts[2]), \
                (verts[2], verts[1]), (verts[1], (self._south_wall[0][0], verts[1][1])), \
                ((self._south_wall[0][0], verts[1][1]), (self._south_wall[0][0], verts[2][1]))]
        cell_6 = [(verts[2], inter3), (inter3, self._obstacles[1][2][1]), \
                (self._obstacles[1][2][1], verts[7]), (verts[7], inter4), \
                (inter4, self._obstacles[2][0][1]), (self._obstacles[2][0][1], verts[2])]
        cell_7 = [(inter2, (self._north_wall[0][0], verts[9][1])), \
                ((self._north_wall[0][0], verts[9][1]), (self._north_wall[0][0], verts[7][1])), \
                ((self._north_wall[0][0], verts[7][1]), verts[7]), (verts[7], inter2)]
        cell_8 = [(inter4, (self._north_wall[0][0], verts[7][1])), \
```

```
                        ((self._north_wall[0][0], verts[7][1]), (self._north_wall[0][0], verts[1][1])), \
                        ((self._north_wall[0][0], verts[1][1]), verts[1]), \
                        self._obstacles[2][2], (self._obstacles[2][1][1], inter4)]
        cell_9 = [((self._south_wall[0][0], verts[1][1]), (self._north_wall[0][0], verts[1][1])), \
                        ((self._north_wall[0][0], verts[1][1]), (self._north_wall[0][0], verts[4][1])), \
                        ((self._north_wall[0][0], verts[4][1]), (self._south_wall[0][0], verts[4][1])), \
                        ((self._south_wall[0][0], verts[4][1]), (self._south_wall[0][0], verts[1][1]))]
        cell_10 = [(verts[4], (self._north_wall[0][0], verts[4][1])), \
                         ((self._north_wall[0][0], verts[4][1]), (self._north_wall[0][0], verts[5][1])), \
                         ((self._north_wall[0][0], verts[5][1]), verts[5]), \
                         self._obstacles[3][1], self._obstacles[3][0]]
        cell_11 = [((self._south_wall[0][0], verts[4][1]), verts[4]), \
                         (verts[4], verts[5]), (verts[5], (self._south_wall[0][0], verts[5][1])), \
                         ((self._south_wall[0][0], verts[5][1]), (self._south_wall[0][0], verts[4][1]))]
        cell_12 = [((self._south_wall[0][0], verts[5][1]), (
```

```
    self._north_wall[0][0], verts[5][1])), \
                ((self._north_wall[0][0], verts[5][1]), (self._north_wall[0][0], verts[3][1])), \
                ((self._north_wall[0][0], verts[3][1]), (self._south_wall[0][0], verts[3][1])), \
                ((self._south_wall[0][0], verts[3][1]), (self._south_wall[0][0], verts[5][1]))]
        cell_13 = [(verts[3], (self._north_wall[0][0], verts[3][1])), \
                 ((self._north_wall[0][0], verts[3][1]), (self._north_wall[0][0], verts[0][1])), \
                 ((self._north_wall[0][0], verts[0][1]), verts[0]), \
                 self._obstacles[4][2], self._obstacles[4][1]]
        cell_14 = [((self._south_wall[0][0], verts[3][1]), verts[3]), \
                 self._obstacles[4][0], self._obstacles[4][3], \
                 (verts[0], (self._south_wall[0][0], verts[0][1])), \
                 ((self._south_wall[0][0], verts[0][1]), (self._south_wall[0][0], verts[3][1]))]
        cell_15 = [((self._south_wall[0][0], verts[0][1]), (self._north_wall[0][0], verts[0][1])), \
                 ((self._north_wall[0][0], verts[0][1]), self._north_wall[1]), \
                 self._east_wall, (self._east_wall[0], (self._south_wall[0][0], verts[0][1]))]
        temp_cells = [cell_0, cell_1, cell_2, cell_3, cell_4
```

```
, cell_5, \
                    cell_6, cell_7, cell_8, cell_9, cell_10, cell_11, \
                    cell_12, cell_13, cell_14, cell_15]
        # create cell objects from the Cell class (final_cells_class.py) given the list of cells above
        concave_north_walled_cells = [3, 4, 6, 14]
        for cell_id in range(len(temp_cells)):
            self._cells[cell_id] = Cell(cell_id, temp_cells[cell_id])
            self._cells[cell_id].setWestBound(self._cells[cell_id].getBoundary()[0])
            if cell_id in concave_north_walled_cells:
                self._cells[cell_id].setEastBound(self._cells[cell_id].getBoundary()[3])
            else:
                self._cells[cell_id].setEastBound(self._cells[cell_id].getBoundary()[2])
            self._cells_left.add(self._cells[cell_id].getCellId())
        return len(self._cells)



    def generateSearchBid(self):
        ''' calculate an agent's bid for a cell given utility calculations
        '''
        # thesis data capture
        self.captureRoundData()
        self._prev_bid = [e for e in self._curr_bid]
        # if agent won its cell in winnerDetermination(),
```

```
submit the same bid again
        if self._submit_same_bid:
            self._submit_same_bid = False
        else:
            # if agent has no viable cells to bid for, submit explicit "no_bid"
            no_cells = self._cells_left.difference(self._cells_in_progress) == self._abandoned_cells
            if no_cells and not self._initial_assign:
                self._curr_bid = [ ]
                self._curr_bid = [AuctionSearch.NOT_BIDDING, Cell.NO_COST]
            else:
                # if agent did not win a cell in winnerDetermination(), generate utilities and a new bid
                self.manager.log_info("Generating a new bid")
                self.generateCellUtilities()
                # if I have cells available, but have lost twice in a row for each one, concede the round.
                if len(self._cell_utilities) == 0:
                    self.manager.log_info("No utility in bidding this round.  Submit explicit no-bid.")
                    bid_cell_id = AuctionSearch.NOT_BIDDING
                    bid_value   = Cell.NO_COST
                elif len(self._cell_utilities) > 1:
                    highest_util = self._cell_utilities[-1][0]
                    second_best  = self._cell_utilities[-2][0]
```

```
            bid_cell_id  = self._cell_utilities[-1][1]
            # allow negative utilities
            if second_best < 0:
                bid_value = self._cells[bid_cell_id].getCost() + highest_util \
                                + second_best + AuctionSearch.EPSILON
            else:
                bid_value = self._cells[bid_cell_id].getCost() + highest_util \
                                - second_best + AuctionSearch.EPSILON
        else:
            bid_cell_id = self._cell_utilities[-1][1]
            highest_util= self._cell_utilities[-1][0]
            bid_value   = self._cells[bid_cell_id].getCost() \
                              + highest_util + AuctionSearch.EPSILON
        if bid_cell_id in self._cells_not_won:
            bid_value += AuctionSearch.EPSILON
        # if my new bid is exactly the same as my previous bid,
        #   and I didn't mean for that to happen (NOT self._submit_same_bid)
        #   increase it by epsilon again. (this occurs very rarely)
        if len(self._prev_bid) > 0 and bid_cell_id == self._prev_bid[0] \
          and bid_value == self._prev_bid[1] and
```

```
                        bid_cell_id != AuctionSearch.NOT_BIDDING:
                    bid_value += AuctionSearch.EPSILON
                self._curr_bid = []
                self._curr_bid.append(bid_cell_id)
                self._curr_bid.append(int(round(bid_value)))
        if self._choose_search_area in [AuctionSearch.BASIC_LIVE_FLY, AuctionSearch.COMPLEX]:
            self.manager.log_info("Utilities calculated and below.  Index[-1] == highest utility cell:")
            self.manager.log_info(self._cell_utilities)
        self.manager.log_info(("My bid: format [ cell , bid ]: ", self._curr_bid))
        if self._curr_bid[0] != AuctionSearch.NOT_BIDDING:
            self.manager.log_info(("bid_cell_cost: ", self._cells[self._curr_bid[0]].getCost()))
        self._bidding_complete = True
        self._round_number += 1



    def generateWaypoints(self, cell_id, start_location):
        ''' Generates, distributes, and prioritizes waypoint objects in a given cell
        @param cell_id: the cell having waypoints created
        @param start_location: the closest corner to last_waypoint from which to start waypoints
        '''
        grid = gps.cartesian_offset(AuctionSearch.AREA_SW_LAT, AuctionSearch.AREA_SW_LON, \
                                    start_location[0], start_location[1])
        chalk_line = []
```

```
        west_bound = self._cells[cell_id].getWestBound()
        east_bound = self._cells[cell_id].getEastBound()
        bot_to_nw  = ro_math.cartesian_distance(grid, west_bound[1])
        bot_to_sw  = ro_math.cartesian_distance(grid, west_bound[0])
        bot_to_ne  = ro_math.cartesian_distance(grid, east_bound[0])
        bot_to_se  = ro_math.cartesian_distance(grid, east_bound[1])
        waypoint_id= 1
        size = 0
        # determine where to start waypoints based on bot-cell relative location
        # if bot closest to east side, start waypoints from east side
        if bot_to_ne < bot_to_nw or bot_to_se < bot_to_sw:
            from_east = True
            # if bot closest to northeast corner, start waypoints from north
            if bot_to_ne < bot_to_se:
                from_north = True
            else:
                from_north = False
        else:
            from_east = False
            # if bot closest to northwest corner, start waypoints from north
            if bot_to_nw < bot_to_sw:
                from_north = True
            else:
                from_north = False
        # if bot is approaching from the east, generate east
```

```
        -to-west sweep line
        if from_east:
            sweep_line = ((east_bound[0][0], east_bound[0][1] - self._sensor_sweep[1]), \
                          (east_bound[1][0], east_bound[1][1] - self._sensor_sweep[1]))
            # generate a "sweeper" line that guides a "chalk_line" for planting waypoints
            sweeper   = [[sweep_line[0][0] + (1.5*self._sensor_sweep[0]), sweep_line[0][1]], \
                          [sweep_line[1][0] - (1.5*self._sensor_sweep[0]), sweep_line[1][1]]]
        # if bot is approaching from the west, generate west-to-east sweep line
        else:
            sweep_line = ((west_bound[0][0], west_bound[0][1] + self._sensor_sweep[1]), \
                          (west_bound[1][0], west_bound[1][1] + self._sensor_sweep[1]))
            # generate a "sweeper" line that guides a "chalk_line" for planting waypoints
            sweeper   = [[sweep_line[0][0] - (1.5*self._sensor_sweep[0]), sweep_line[0][1]], \
                          [sweep_line[1][0] + (1.5*self._sensor_sweep[0]), sweep_line[1][1]]]
        # if bot is nearest the north, lay waypoints from north to south to start "lawnmower" pattern
        if from_north:
            inflection = 1
```

```
            else:
                inflection = 0
            # sweep a vertical "chalk_line" across the cell as a guide for planting waypoints
            if from_east:
                while sweeper[0][1] >= west_bound[0][1]:
                    for bound in self._cells[cell_id].getBoundary():
                        intersection = ro_math.segment_intersect(bound[0], bound[1], \
                                                                                                                    sweeper[0],
                                                                                                                    sweeper[1])
                        if intersection != None:
                            chalk_line.append(intersection)
                    # place waypoints at specified intervals along the chalk_line
                    if len(chalk_line) < 2:
                        break
                    y = chalk_line[0][1]
                    max_x = chalk_line[0][0] - self._sensor_sweep[1]
                    min_x = chalk_line[1][0] + self._sensor_sweep[1]
                    inverter = divmod(inflection, 2)[1]
                    # place waypoints from south to north
                    if inverter == 0:
                        x = min_x
                        while x <= max_x:
                            waypoint = Waypoint(self._cells[
```

```
                    cell_id].getCellId(), waypoint_id
                    , \
                                        (x, y), (
                                            AuctionSearch
                                            .AREA_SW_LAT,

                                            AuctionSearch
                                            .AREA_SW_LON)
                                            )
                self._cells[cell_id].addWaypoint(
                    waypoint)
                x += self._sensor_sweep[1]
                size += self._sensor_sweep[1]
                waypoint_id += 1
        # place waypoints from north to south
        elif inverter == 1:
            x = max_x
            while x >= min_x:
                waypoint = Waypoint(self._cells[
                    cell_id].getCellId(), waypoint_id
                    , (x, y), \
                                                (
                                                    AuctionSearch
                                                    .
                                                    AREA_SW_LAT
                                                    ,
                                                    AuctionSearch
                                                    .
                                                    AREA_SW_LON
                                                    ))
                self._cells[cell_id].addWaypoint(
                    waypoint)
                x -= self._sensor_sweep[1]
```

```
                    size += self._sensor_sweep[1]
                    waypoint_id += 1
                chalk_line = [ ]
                inflection += 1
                size += self._sensor_sweep[0]
                # move the chalk-line waypoint guide to the left by one sensor-sweep (east to west)
                sweeper[0][1] -= self._sensor_sweep[0]
                sweeper[1][1] -= self._sensor_sweep[0]
        else:
            while sweeper[0][1] <= east_bound[0][1]:
                for bound in self._cells[cell_id].getBoundary():
                    intersection = ro_math.segment_intersect(bound[0], bound[1], \
                                                                                sweeper[0],
                                                                                sweeper[1])
                    if intersection != None:
                        chalk_line.append(intersection)
                # place waypoints at specified intervals along the chalk_line
                if len(chalk_line) < 2:
                    break
                y = chalk_line[0][1]
                max_x = chalk_line[0][0] - self._sensor_sweep[1]
                min_x = chalk_line[1][0] + self._sensor_sweep[1]
                inverter = divmod(inflection, 2)[1]
```

```
        # place waypoints from south to north
        if inverter == 0:
            x = min_x
            while x <= max_x:
                waypoint = Waypoint(self._cells[cell_id].getCellId(), waypoint_id, (x, y), \
                                    (AuctionSearch.AREA_SW_LAT, AuctionSearch.AREA_SW_LON))
                self._cells[cell_id].addWaypoint(waypoint)
                x += self._sensor_sweep[1]
                size += self._sensor_sweep[1]
                waypoint_id += 1
        # place waypoints from north to south
        elif inverter == 1:
            x = max_x
            while x >= min_x:
                waypoint = Waypoint(self._cells[cell_id].getCellId(), waypoint_id, (x, y), \
                                    (AuctionSearch.AREA_SW_LAT, AuctionSearch.AREA_SW_LON))
                self._cells[cell_id].addWaypoint(waypoint)
                x -= self._sensor_sweep[1]
                size += self._sensor_sweep[1]
```

```
                        waypoint_id += 1
                    chalk_line  = [ ]
                    inflection += 1
                    # move the chalk-line waypoint guide to the right by one sensor-sweep (west to east)
                    sweeper[0][1] += self._sensor_sweep[0]
                    sweeper[1][1] += self._sensor_sweep[0]
                    size += self._sensor_sweep[0]
            self._cells[cell_id].setSize(size)


    def getInTheAuction(self):
        ''' start an auction and reinitialize all associated data structures
        '''
        self.manager.log_warn("Auction start message received")
        self._agent.setSearchAuction()
        self._data_auction_time.append(rospy.Time.now())
        self._round_number     = 0
        self._inbound_bids     = [ ]
        self._inbound_statuses= [ ]
        self._search_roll_call.clear()
        self._bid_roll_call.clear()
        self._round_tracker.clear()
        self._complete_roll_call.clear()
        self._auction_started = True
        self._mid_search_bid  = False
        self._auction_complete= False
        self._bidding_complete= False
        self._bids_updated     = False
        self._cell_update_complete = False

```

```python


    def internalUpdateCells(self):
        ''' update local cell knowledge given winning bids
            from an auction
        '''
        for agent_key in self._all_bids:
            for cell_key, cost in self._all_bids[agent_key].
                items():
                if cell_key in self._cells:
                    # if owning agent is no longer alive,
                        abort assignment
                    if agent_key not in self.manager.
                        subswarm_keys:
                        self.revertCell(cell_key)
                        self._cells_in_progress.discard(
                            cell_key)
                        self._cells_left.add(cell_key)
                    else:
                        # if another agent won a cell I'm
                            assigned to, remove assignment
                        if cell_key in self._agent.
                            getMyCellIds() and \
                           agent_key != self._agent.
                              getSearcherId():
                            if cell_key == self._agent.
                                getCurrCellId():
                                self._agent.
                                    resetCurrWaypointId()
                            self.removeCellAssignment(
                                cell_key)
                        self._cells[cell_key].setStatus(Cell
                            .ASSIGNED)
```

```
                        self._cells[cell_key].setOwner(agent_key)
                        self._cells[cell_key].setCost(cost)
                        self._cells_changed.add(cell_key)



    def makeCellActive(self):
        ''' set an assigned cell to in-progress once a waypoint has been reached
        '''
        my_cell = self._agent.getCurrCellId()
        if my_cell != None:
            if self._cells[my_cell].getStatus() != Cell.IN_PROGRESS:
                self.manager.log_warn("this is where I set cell %d to IN_PROGRESS" % my_cell)
                self._cells[my_cell].setStatus(Cell.IN_PROGRESS)
                # thesis data capture line
                if len(self._data_robot_searching) == 0:
                    self._data_robot_searching.append(rospy.Time.now())
                    self._am_searching = True
                self._cells_changed.add(my_cell)
                self._cells_in_progress.add(my_cell)



    def moveToNextCell(self):
        ''' move to an assigned cell upon completion of in-progress cell
        '''
```

```
self.manager.log_info("finished cell %d, moving to cell %d." \
                      % (self._agent.getCurrCellId(), self._agent.getMyCellIds()[1]))
bot = self.manager.get_own_state().state.pose.pose.position
self._been_there.add(self._agent.getCurrCellId())
self._cells_in_progress.discard(self._agent.getCurrCellId())
self._cells_left.discard(self._agent.getCurrCellId())
# getCurrCellId() is now the next cell in getMyCellIds()
self._agent.removeCell(self._agent.getCurrCellId())
self._agent.resetCurrWaypointId()
self._cells_changed.add(self._agent.getCurrCellId())
self._cells[self._agent.getCurrCellId()].deleteWaypoints()
self.generateWaypoints(self._agent.getCurrCellId(), (bot.lat, bot.lon))
self._cells[self._agent.getCurrCellId()].setStatus(Cell.IN_PROGRESS)
# thesis data capture line
if len(self._data_robot_searching) == 0:
    self._data_robot_searching.append(rospy.Time.now())
    self._am_searching = True
self._cells_changed.add(self._agent.getCurrCellId())
self._cells_in_progress.add(self._agent.getCurrCellId())


```

```
    def reassignCell(self, cell_id, agent_id, agent_bid):
        ''' change assignment of a cell from one agent to
            another
        @param cell_id: the id of the cell being reassigned
        @param agent_id: the id of the agent being
            reassigned the cell
        @param agent_bid: the bid amount that agent_id won
            cell_id for
        '''
        if cell_id in self._agent.getMyCellIds():
            self._cells[cell_id].deleteWaypoints()
            self._agent.removeCell(cell_id)
            self._cells_changed.add(cell_id)
            self._cells[cell_id].setStatus(Cell.ASSIGNED)
            self._cells[cell_id].setOwner(agent_id)
            if agent_bid > self._cells[cell_id].getCost():
                self._cells[cell_id].setCost(agent_bid)
            self._cells_changed.add(cell_id)
        else:
            self.manager.log_info("ReassignCell() failed: 
                cell_id not in MyCellIds()")



    def removeCellAssignment(self, cell_id):
        ''' change cell status to assignment-removed so
            other agents can detect it
        @param cell_id: the id of the cell being
            disassociated
        '''
        if cell_id in self._agent.getMyCellIds():
            self._cells[cell_id].deleteWaypoints()
```

```
                self._agent.removeCell(cell_id)
                self.revertCell(cell_id)



    def revertCell(self, cell_id):
        ''' change cell status from assignment-removed to available
        Setting a cell to "assignment_removed" allows other agents to
        detect that an "assigned" cell should now be considered "available."
        @param cell_id: the id of the cell being set to available
        '''
        self._cells[cell_id].setOwner(Cell.NO_OWNER)
        self._cells[cell_id].setCost(Cell.NO_COST)
        self._cells_in_progress.discard(cell_id)
        if self._cells[cell_id].getStatus() != Cell.ASSIGNMENT_REMOVED:
            self._cells[cell_id].setStatus(Cell.ASSIGNMENT_REMOVED)
            self._cells_changed.add(cell_id)
        else:
            self._cells[cell_id].setStatus(Cell.AVAILABLE)
            self._cells_changed.add(cell_id)



    def sendAuctionComplete(self):
        ''' send a single message telling other agents that agent is finished with auction
        '''
```

```
            parser = bytes.UShortParser()
            parser.value = self._agent.getSearcherId()
            report = self.manager.behavior_data_msg
            report.id = bytes.AUCTION_COMPLETE
            report.params = parser.pack()
            self.manager.behavior_data_publisher.publish(report)



    def setWaypoint(self, waypoint_loc):
        ''' send a speed waypoint command message with lat/
            lon/alt/speed information
        @param waypoint_loc: the waypoint to set autopilot
            to
        '''
        if not self._loiter_wait:
            self._loiter_checkpoint = waypoint_loc
        self.manager.spd_wp_cmd_msg.lat   = waypoint_loc[0]
        self.manager.spd_wp_cmd_msg.lon   = waypoint_loc[1]
        self.manager.spd_wp_cmd_msg.alt   = self.manager.
            ap_wpt.z
        self.manager.spd_wp_cmd_msg.speed = self._agent.
            getSpeed()



    def shareAuctionComplete(self, num_agents):
        ''' lossy-comms tolerant way to reliably communicate
              auction status with agents
        @param num_agents: the number of agents in the
            subswarm executing AuctionSearch
        '''
        if self._auc_msg_count < AuctionSearch.MESSAGE_COUNT
```

```
            :
            if self._auc_msg_count == 0:
                self._complete_roll_call.clear()
            if divmod(self._auc_msg_count, 3)[1] == 0:
                self.sendAuctionComplete()
            self._auc_msg_count += 1
        # If searcher has not heard from all others, request auction status from them
        elif self._auc_msg_count >= AuctionSearch.MESSAGE_COUNT and \
          len(self._complete_roll_call) < (num_agents - 1):
            if divmod(self._auc_msg_count, 5)[1] == 0:
                self.auctionCompleteRequest()
                self.manager.log_info("requesting complete statuses, auction %d" % self._auction_number)
            self._auc_msg_count += 1
        # If searcher has heard from all other active agents, finish auction
        if len(self._complete_roll_call) >= (num_agents - 1) and \
          self._auc_msg_count >= AuctionSearch.MESSAGE_COUNT:
            self._auc_msg_count = 0
            self._auction_complete = True
            self._winners_picked = False
            # capture thesis data
            self.captureRoundData()
            self.manager.log_info("- - - - auction is complete - - - - ")
```

```python
    def shareBids(self, num_agents):
        ''' lossy-comms tolerant way to reliably communicate bids with agents
        '''
        if self._bidding_complete:
            if self._bid_msg_count < AuctionSearch.MESSAGE_COUNT:
                if divmod(self._bid_msg_count, 3)[1] == 0:
                    self.bidStatusUpdate(False)
                self._bid_msg_count += 1
            # If searcher has not heard from all others, request status from them
            elif self._bid_msg_count >= AuctionSearch.MESSAGE_COUNT and \
              len(self._bid_roll_call) < (num_agents - 1):
                if divmod(self._bid_msg_count, 3)[1] == 0:
                    self.bidStatusRequest()
                    self.manager.log_info("requesting bids.  round %d" % self._round_number)
                self._bid_msg_count += 1
            # If searcher has heard from all other active agents, report ready status
            if len(self._bid_roll_call) >= (num_agents - 1) and \
              self._bid_msg_count >= AuctionSearch.MESSAGE_COUNT:
                self._bid_msg_count = 0
                self._bid_roll_call.clear()
                self._bids_updated = True
                self._round_tracker.clear()
                self.manager.log_info("- - - - bid update complete - - - - ")
                self.manager.log_info("")
```

```python
    def shareStatuses(self, num_agents):
        ''' lossy-comms tolerant way to reliably communicate cell statuses with agents
        @param num_agents: the number of agents in the subswarm executing AuctionSearch
        '''
        if self._message_count < AuctionSearch.MESSAGE_COUNT:
            if divmod(self._message_count, 3)[1] == 0:
                self.cellStatusUpdate()
            self._message_count += 1
        # If searcher has not heard from all others, request status from them
        elif self._message_count >= AuctionSearch.MESSAGE_COUNT and \
          len(self._search_roll_call) < (num_agents - 1):
            if divmod(self._message_count, 5)[1] == 0:
                self.cellStatusRequest()
                self.manager.log_info("requesting cell statuses, auction %d, round %d" \
                                      % (self._auction_number, self._round_number))
            self._message_count += 1
        # If searcher has heard from all other active agents, update cells
        if len(self._search_roll_call) >= (num_agents - 1) and \
          self._message_count >= AuctionSearch.MESSAGE_COUNT
```

```
                :
          self.externalUpdateMyCells()
          self._message_count = 0
          self._cell_update_complete = True
          self.manager.log_info("- - - - cell update complete - - - - ")



    def startAuction(self, next_cell_claimed):
        ''' send a burst of auction start messages to other agents
        '''
        parser = bytes.NewAuctionParser()
        parser.source_id = self._agent.getSearcherId()
        if next_cell_claimed:
            parser.next_cell_id = self._agent.getCurrCellId()
        else:
            parser.next_cell_id = AuctionSearch.NOT_BIDDING
        parser.auction_number = self._auction_number
        parser.search_auction = True
        self.claim_next_cell  = next_cell_claimed
        for i in range(AuctionSearch.MESSAGE_COUNT):
            if i == 0:
                self.manager.log_warn("sending auction start message.")

            report = self.manager.behavior_data_msg
            report.id = bytes.AUCTION_NEW
            report.params = parser.pack()
            self.manager.behavior_data_publisher.publish(report)
```

```
        self._start_auction = False
        self._auction_complete = False
        self.getInTheAuction()



    def stayInMyCell(self):
        ''' command agent to loiter at last waypoint after finishing its cell
        '''
        wait_for_cell = False
        if len(self._agent.getMyCellIds()) > 1:
            self.manager.log_info("finished my cell, but am waiting for a potentially better next cell.")
        else:
            self.manager.log_info("finished my last cell.  Standing by.")
        if self._agent.getCurrCellId() != None:
            self._been_there.add(self._agent.getCurrCellId())
        self._cells_in_progress.discard(self._agent.getCurrCellId())
        self._cells_left.discard(self._agent.getCurrCellId())
        # so CurrCellId() now corresponds to my next cell, if any
        self._agent.removeCell(self._agent.getCurrCellId())
        self._agent.resetCurrWaypointId()
        if self._agent.getCurrCellId() != None:
            self._cells_changed.add(self._agent.getCurrCellId())
        self._loiter_wait = True

```

```
    def submitSearchBid(self, searcher_id, cell_id, bid_value):
        ''' send a single message telling other agents bid information
        @param searcher_id: id of the searcher submitting the bid
        @param cell_id: id of the cell searcher_id bid for
        @param bid_value: amount that searcher_id bids for cell_id
        '''
        parser = bytes.AuctionSearchBidParser()
        parser.my_id   = searcher_id
        parser.cell_id = cell_id
        parser.bid_val = bid_value
        report = self.manager.behavior_data_msg
        report.id = bytes.AUCTION_BID
        report.params = parser.pack()
        self.manager.behavior_data_publisher.publish(report)



    def syncRounds(self, num_agents):
        ''' check whether all agents are in the same round, behind, or ahead in an auction
        @param num_agents: the number of agents in the subswarm executing AuctionSearch
        '''
        synced = False
        self._round_tracker.add(self._round_number)
        diff_round_nums = len(self._round_tracker)
        # if all agents are in the same round, length of
```

```
            this set will be 1,
        #   so return true because agents are synced.
        if diff_round_nums == 0 or diff_round_nums == 1:
            synced = True
        else:
            if diff_round_nums == 0:
                max_round_num = 0
            else:
                max_round_num = max(self._round_tracker)
        # if there is more than one number in self.
            _round_tracker set, then agents are out of sync
        if diff_round_nums > 1:
            # if my round number is the same as max, I am
                ahead of other agents and need to wait.
            if self._round_number == max_round_num:
                min_num = min(self._round_tracker)
                if max_round_num - min_num > 2:
                    synced = True
                elif divmod(self._sync_msg_count, 3)[1] ==
                    0:
                    self._round_tracker.clear()
                    self.auctionStatusRequest()
                    self.manager.log_info("I'm ahead.  
                        Requesting round numbers.")
                    synced = False
                self._sync_msg_count += 1
            # if my round number is not the same as max, I
                need to continue in order to catch up.
            else:
                synced = True
        if synced:
            self._sync_msg_count = 0
        return synced
```

```



    def testWaypoint(self, waypoint_loc):
        ''' check whether an agent has arrived at a specified waypoint
        @param waypoint_loc: the waypoint agent is traveling toward
        '''
        if not self._initial_assign:
            # if agent is at waypoint, go to next waypoint
            if not self._loiter_wait:
                bot = self.manager.get_own_state().state.pose.pose.position
                dist_to_wp = gps.gps_distance(waypoint_loc[0], waypoint_loc[1], bot.lat, bot.lon)
                if dist_to_wp < AuctionSearch.CAPTURE_DIST and self._agent.getCurrCellId() != None:
                    if self._cells[self._agent.getCurrCellId()].getStatus() == Cell.IN_PROGRESS:
                        self._agent.incrementCurrWaypointId()
                    elif self._agent.getCurrWaypointId() == 0 and not self._agent._IS_SEARCH_AUCTION \
                      and self._cells[self._agent.getCurrCellId()].getStatus() == Cell.ASSIGNED:
                        self._agent.incrementCurrWaypointId()



```

```
    def winnerDetermination(self):
        ''' determine highest bidder from a set of bids and direct auction termination
        '''
        winner_id   = self._agent.getSearcherId()
        highest_bid = 0.0
        self._same_bids = True # if all agents submit the same bids a second time, all are happy.
        # consolidate all bid information from others and myself
        self.consolidateBids()
        self.displayShortReport()
        # if all agents are happy and no conflicts remain, commit to assignments.
        if self._same_bids:
            if not self._auction_complete:
                if self._curr_bid[0] != AuctionSearch.NOT_BIDDING:
                    self.generateCellAssignment()
                self.internalUpdateCells()
                if not self._initial_assign:
                    self._mid_search_bid = True
            self._cell_update_complete = False
            self._search_roll_call.clear()
            self._submit_same_bid = False
            self._winners_picked = True
            self.manager.log_info("SAME_BIDS == TRUE, and no cell conflicts.  Committing assignments.")
        # if agents are not happy, determine who's bid won
        else:
            winner_id   = self._agent.getSearcherId()
            highest_bid = self._curr_bid[1] # assume agent submitted highest bid unless proven otherwise
```

```
        cell_key     = self._curr_bid[0]
        for agent_key in self._all_bids:
            # if another agent placed a bid for the same cell as me, highest bid is winner
            if cell_key in self._all_bids[agent_key] and agent_key != self._agent.getSearcherId() \
              and cell_key != AuctionSearch.NOT_BIDDING:
                agent_bid = self._all_bids[agent_key][cell_key]
                if self._all_bids[agent_key][cell_key] > highest_bid:
                    highest_bid = agent_bid
                    winner_id   = agent_key
                    # if another agent bid higher for a cell I am assigned, I relinquish it
                    if cell_key in self._agent.getMyCellIds():
                        if cell_key == self._agent.getCurrCellId():
                            self._agent.resetCurrWaypointId()
                        self.reassignCell(cell_key, agent_key, agent_bid)
                        self._cells_changed.add(cell_key)
                if self._all_bids[agent_key][cell_key] == highest_bid:
                    if winner_id < agent_key:
                        highest_bid = self._all_bids[agent_key][cell_key]
                        winner_id   = agent_key
```

```
                if cell_key in self._agent.getMyCellIds():
                    if cell_key == self._agent.getCurrCellId():
                        self._agent.resetCurrWaypointId()
                    self.reassignCell(cell_key, agent_key, agent_bid)
                    self._cells_changed.add(cell_key)
# if I won the cell I bid for this round, submit the same bid next round
if winner_id == self._agent.getSearcherId():
    if cell_key != AuctionSearch.NOT_BIDDING:
        self._submit_same_bid = True
        if self._curr_bid[1] > self._cells[cell_key].getCost():
            self._cells[cell_key].setCost(self._curr_bid[1])
        self._cells_changed.add(cell_key)
# I did not win the cell I bid for
else:
    # if I've lost the same cell multiple times, conclude I won't win it
    if cell_key in self._cells_not_won and cell_key != AuctionSearch.NOT_BIDDING:
        self.manager.log_info("cell %d is outside of threshold. Abandon pursuit for other cells." \
                              % cell_key)
        self._abandoned_cells.add(cell_key)
        if highest_bid > self._cells[cell_key].getCost():
```

```
                    self._cells[cell_key].setCost(highest_bid)
            # if I lost the round, keep track of the cell_id
            else:
                self.manager.log_info("I lost, but keeping track of cell %d." % cell_key)
                self._cells_not_won.add(cell_key)
                if highest_bid > self._cells[cell_key].getCost():
                    self._cells[cell_key].setCost(highest_bid)
                self._cells_changed.add(cell_key)
        self._inbound_bids = [ ]
        self._bids_updated = False
        self._bidding_complete = False



class Cell(object):
    '''
    Class for maintaining attributes of each cell created by decomposition.
    Each Cell represents a biddable resource in Auctions.
    '''
    # CELL status enumerations
    AVAILABLE    = 0
    ASSIGNED     = 1
    IN_PROGRESS  = 2
    ASSIGNMENT_REMOVED = 3
    COMPLETE     = 4

```

```
    # other enumerations
    NO_OWNER      = 255
    NO_COST       = 0.0
    PRIVATE_VALUE = 8000

    def __init__(self, cell_id, boundary):
        self._cell_id    = cell_id
        self._boundary   = boundary
        self._west_bound= [ ]
        self._east_bound= [ ]
        self._waypoints = [ ]
        self._neighbors = [ ]
        self._size       = 0
        self._utility    = Cell.NO_COST
        self._bid_amts   = Cell.NO_COST
        self._cost       = Cell.NO_COST
        self._status     = Cell.AVAILABLE
        self._owner      = Cell.NO_OWNER
        self._private_val = Cell.PRIVATE_VALUE

    def getCellId(self):
        return self._cell_id

    def getBoundary(self):
        return self._boundary

    def getBoundaryGrids(self):
        boundary_grids = [ ]
        for bound in self.getBoundary():
            boundary_grids.append(bound[0])
            boundary_grids.append(bound[1])
        return boundary_grids

```

```
    def getWaypoints(self):
        return self._waypoints

    def getNeighbors(self):
        return self._neighbors

    def getSize(self):
        return self._size

    def getStatus(self):
        return self._status

    def getOwner(self):
        return self._owner

    def getBidAmounts(self):
        return self._bid_amts

    def getCost(self):
        return self._cost

    def getValue(self):
        return self._private_val

    def getUtility(self):
        return self._utility

    def getWestBound(self):
        return self._west_bound

    def getEastBound(self):
        return self._east_bound

```

```
    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 getWaypointIds(self):
        waypoint_ids = [ ]
        for waypoint in self._waypoints:
            waypoint_ids.append(waypoint.getWaypointId())
        return waypoint_ids

    def addWaypoint(self, waypoint):
        self._waypoints.append(waypoint)

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

    def addNeighbor(self, neighbor_id):
        self._neighbors.append(neighbor_id)

    def deleteNeighbor(self, neighbor_id):
        self._neighbors.remove(neighbor)

    def deleteBidAmounts(self):
        self._bid_amts = Cell.NO_COST
```

```python

    def setBidAmounts(self, amt):
        self._bid_amts = amt

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

    def setStatus(self, new_status):
        self._status = new_status

    def setOwner(self, owner_id):
        self._owner = owner_id

    def setCost(self, price):
        self._cost = price

    def setValue(self, value):
        self._private_val = value

    def setUtility(self, util):
        self._utility = util

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

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



class Waypoint(object):
    '''
    Class for maintaining attributes of each waypoint within
```

```
        a cell
    '''
    def __init__(self, cell_id, waypoint_id, location,
        sw_corner):
        self._waypoint_id = waypoint_id
        self._cart_loc    = location
        self._lat_lon_loc = gps.gps_offset(sw_corner[0],
            sw_corner[1], \
                                           location[1],
                                              location[0])

    def getWaypointId(self):
        return self._waypoint_id

    def getCartesianLocation(self):
        return self._cart_loc

    def getLatLonLocation(self):
        return self._lat_lon_loc



class Searcher(object):
    '''
    Class for maintaining attributes of each searcher
    '''
    def __init__(self, searcher_id):
        self._searcher_id    = searcher_id
        self._IS_SEARCHER    = True
        self._IS_SEARCH_AUCTION = True
        self._curr_waypoint = 0
        self._speed = 0
        self._endurance = 0.0
```

```
        self._my_cell_ids    = [ ]
        self._been_there     = [ ]

    def getSearcherId(self):
        return self._searcher_id

    def getMyCellIds(self):
        return self._my_cell_ids

    def getCurrCellId(self):
        if len(self._my_cell_ids) > 0:
            return self._my_cell_ids[0]
        else:
            return None

    def getCurrWaypointId(self):
        return self._curr_waypoint

    def getEndurance(self):
        return self._endurance

    def getSpeed(self):
        return self._speed

    def setSearchAuction(self):
        self._IS_SEARCH_AUCTION = True

    def setEndurance(self, endurance):
        self._endurance = endurance

    def setSpeed(self, speed):
        self._speed = speed

```

```
    def addCell(self, cell_id):
        self._my_cell_ids.append(cell_id)

    def removeCell(self, cell_id):
        if cell_id in self._my_cell_ids:
            self._my_cell_ids.remove(cell_id)

    def removeAssignedCells(self):
        curr_cell = self.getCurrCellId()
        if curr_cell == None:
            self._my_cell_ids = [ ]
        else:
            self._my_cell_ids = [curr_cell]

    def removeAllAssignments(self):
        self._my_cell_ids = [ ]

    def incrementCurrWaypointId(self):
        self._curr_waypoint += 1

    def resetCurrWaypointId(self):
        self._curr_waypoint = 0
```

THIS PAGE INTENTIONALLY LEFT BLANK

List of References

[1] J. Bellingham, M. Tillerson, A. Richards, and J. How, *Multi-Task Allocation and Path Planning for Cooperating UAVs*. New York City, NY, USA: Springer, 2003, ch. 1, pp. 23–41.

[2] P. Scharre, "Robotics on the battlefield part ii: The coming swarm," Washington, DC, USA, 2014. Available: https://www.cnas.org/publications/reports/robotics-on-the-battlefield-part-ii-the-coming-swarm

[3] M. Brambilla, E. Ferrante, M. Birattari, and M. Dorigo, "Swarm robotics: A review from the swarm engineering perspective," *Swarm Intelligence*, vol. 7, no. 1, pp. 1–41, 2013, doi: 10.1007/s11721-012-0075-2.

[4] A. Burkle, F. Segor, and M. Kollmann, "Towards autonomous micro uav swarms," *Journal on Intelligent Robotic Systems*, vol. 61, pp. 339–353, 2011, doi: 10.1007/s10846-010-9492-x.

[5] M. Dias, R. Zlot, N. Kalra, and A. Stentz, "Market-based multirobot coordination: A survey and analysis," in *Proceedings of the IEEE*, no. 7, 2006, vol. 94, pp. 1257–1270, doi: 10.1109/JPROC.2006.876939.

[6] S. Edwards, "Swarming and the future of warfare," Ph.D. dissertation, Public Policy Analysis, Pardee RAND Graduate School, Santa Monica, CA, USA, 2005.

[7] P. Sujit and R. Beard, "Distributed sequential auctions for multiple uav task allocation," in *Proceedings of the American Control Conference*, New York City, NY, 2007, pp. 3955–3960, doi: 10.1109/ACC.2007.4282558.

[8] J. McLurkin, "Stupid robot tricks: A behavior-based distributed algorithm library for programming swarms of robots," M.S. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 2004.

[9] G. Vasarhelyi, C. Viragh, G. Somorjai, N. Tarcai, T. Szorenyi, T. Nepusz, and T. Vicsek, "Outdoor flocking and formation flight with autonomous aerial robots," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, no. 1, 2014, vol. 7, pp. 3866–3873, doi: 10.1109/IROS.2014.6943105.

[10] A. Stranieri, E. Ferrante, A. Turgut, V. Trianni, C. Pinciroli, M. Birattari, and M. Dorigo, "Self-organized flocking with a heterogeneous mobile robot swarm," Universite Libre De Bruxelles, Bruxelles, Belgium, Tech. Rep. TR/IRIDIA/2011-012, apr 2011.

[11] L. Hunsaker. (2015). ARSENL reaches its ultimate goal of 50 autonomous UAVs in flight. [Online]. Available: https://my.nps.edu/-/arsenl-reaches-its-ultimate-goal-of-50-autonomous-uavs-in-flig-1

[12] M. Rubenstein, A. Christian, and N. Radhika, “Kilobot: A low cost scalable robot system for collective behaviors,” in *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 3293–3298, doi: 10.1109/ICRA.2012.6224638.

[13] M. Senanayake, I. Senthooran, J. Barca, H. Chung, J. Kamruzzaman, and M. Murshed, “Search and tracking algorithms for swarms of robots: A survey,” *Robotics and Autonomous Systems*, vol. 75, no. 1, pp. 422–434, 2015, doi: 10.1016/j.robot.2015.08.010.

[14] B. Gerkey and M. Mataric, “Sold!: Auction methods for multirobot coordination,” in *IEEE Transactions on Robotics and Automation*, no. 5, 2002, vol. 18, pp. 758–768, doi: 10.1109/TRA.2002.803462.

[15] H. Choset and et al, *Principles of Robot Motion: Theory, Algorithms, and Implementation*, 1st ed. Massachusetts Institute of Technology, Cambridge, MA, USA: MIT Press, 2005.

[16] T. Chung, G. Hollinger, and V. Isler, “Search and pursuit-evasion in mobile robotics: A survey,” *Autonomous Robots*, vol. 31, no. 4, pp. 299–316, 2011, doi: 10.1007/s10514-011-9241-4.

[17] T. Chung and T. Stevens, “Autonomous search and counter-targeting using levy search models,” in *2013 IEEE International Conference on Robotics and Automation*, 2013, pp. 3953–3960.

[18] T. Chung and J. Burdick, “A decision-making framework for control strategies in probabilistic search,” in *2007 IEEE International Conference on Robotics and Automation*, 2007, pp. 4386–4393, doi: 10.1109/ROBOT.2007.364155.

[19] P. Almeida, G. Goncalves, and J. Sousa, “Multi-uav platform for integration in mixed-initiative coordinated missions,” in *First IFAC Workshop on Multivehicle Systems*, 2006, vol. 1, pp. 70–75, doi: 10.3182/20061002-2-BR-4906.00013.

[20] D. Dionne and C. Rabbath, “Multi-uav decentralized task allocation with intermittent communications: The dtc algorithm,” in *Proceedings of the 2007 American Control Conference*, 2007, vol. 26, pp. 5406–5411, doi: 10.1109/ACC.2007.4282637.

[21] T. Stirling, J. Roberts, J.-C. Zufferey, and D. Floreano, “Indoor navigation with a swarm of flying robots,” in *International Conference on Robotics and Automation*, 2012, vol. 3, pp. 4641–4647, doi: 10.1109/ICRA.2012.6224987.

[22] D. Lau, "Investigation of coordination algorithms for swarm robotics conducting area search," M.S. thesis, Graduate School of Operations and Information Sciences, Naval Postgraduate School, 2015.

[23] T. Chung, M. Kress, and J. Royset, "Probabilistic search optimization and mission assignment for heterogeneous autonomous agents," in *2009 IEEE International Conference on Robotics and Automation*, 2009, pp. 939–945, doi: 10.1109/ROBOT.2009.5152215.

[24] T. Shima, S. .Rasmussen, and P. Chandler, "Uav team decision and control using efficient collaborative estimation," in *2005 American Control Conference*, no. 129, 2005, vol. 6, pp. 4107–4112, doi: 10.1109/ACC.2005.1470621.

[25] Y. Jin, A. Minai, and M. Polycarpou, "Cooperative real-time search and task allocation in uav teams," in *Proceedings of the 42nd IEEE Conference on Decision and Control*, 2003, vol. 1, pp. 7–12, doi: 10.1109/CDC.2003.1272527.

[26] 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.

[27] 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.

[28] L. Brunet, H. Choi, and J. How, "Consensus-based auction approaches for decentralized task assignment," in *American Institute of Aeronautics and Astronautics Guidance, Navigation and Control Conference and Exhibit*, 2008, vol. 1, pp. 1–24, doi: 10.2514/6.2008-6839.

[29] M. Day, "Multi-agent task negotiation among uavs to defend against swarm attacks," M.S. thesis, Graduate School of Operational and Information Sciences, Naval Postgraduate School, Monterey, CA, 2012.

[30] L. Hunsberger and B. Grosz, "A combinatorial auction for collaborative planning," in *International Conference on Multi-Agent Systems*, no. 1, 2000, vol. 4, pp. 151–158, 10.1109/ICMAS.2000.858447.

[31] R. Olfati-Saber and R. Murray, "Consensus problems in networks of agents with switching topology and time-delays," *IEEE Transactions on Automatic Control*, vol. 49, no. 9, 2004.

[32] M. Berhault, H. Huang, P. Keskinocak, S. Koenig, W. Elmaghraby, P. Griffin, and A. Kleywegt, "Robot exploration with combinatorial auctions," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, no. 1, 2003, vol. 2, pp. 1957–1962, doi: 10.1109/IROS.2003.1248932.

[33] M. Lagoudakis, M. Berhault, S. Koenig, P. Keskinocak, and A. Kleywegt, "Simple auctions with performance guarantees for multi-robot task allocation," in *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, no. 1, 2004, vol. 1, pp. 698–705, doi: 10.1109/IROS.2004.1389434.

[34] S. Parsons, J. Rodriguez-Aguilar, and M. Klein, "Auctions and bidding: A guide for computer scientists," *ACM Computing Surveys*, vol. 43, no. 10, 2011, doi: 10.1.1.332.1259.

[35] A. Kwasnica, J. Ledyard, D. Porter, and C. DeMartini, "A new and improved design for multiobject iterative auctions," *Management Science*, vol. 51, no. 3, pp. 419–434, 2005, doi: 10.1287/mnsc.1040.0334.

[36] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall, 2010.

[37] M. Alighanbari and J. How, "Decentralized task assignment for unmanned aerial vehicles," in *IEEE Conference on Decision and Control, European Control Conference*, no. 1, 2005, vol. 44, pp. 5669–5673, doi: 10.1109/CDC.2005.1583066.

[38] S. Sariel and T. Balch, "Real time auction based allocation of tasks for multi-robot exploration problem in dynamic environments," in *National Conference on Artificial Intelligence*, 2005.

[39] G. Prasad, A. Prasad, and S. Rao, "A combinatorial auction mechanism for multiple resource procurement in cloud computing," *IEEE Transactions on Cloud Computing*, vol. 3, 2016, doi: 10.1109/TCC.2016.2541150.

[40] G. Zhu, S. Sangwan, and T. Lu, "Mechanism design of online multi-attribute reverse auction," in *Hawaii International Conference on Systems Sciences*, 2009, vol. 42, pp. 1–7, doi: 10.1109/HICSS.2009.306.

[41] P. Milgrom, "Putting auction theory to work: The simultaneous ascending auction," *Journal of Political Economy*, vol. 108, no. 2, pp. 245–272, 1999, doi: 10.1086/262118.

[42] K. Sherstyuk, "Complexity and bidder behavior in iterative auctions," in *Economics Bulletin*, no. 4, 2011, vol. 31, pp. 2769–2776.

[43] S. Vries and R. Vohra, "Combinatorial auctions: A survey," *Informs Journal on Computing*, vol. 15, no. 3, pp. 284–309, 2003, doi: 10.1287/ijoc.15.3.284.16077.

[44] T. Sandholm, S. Suri, A. Gilpin, and D. Levine, "Winner determination in combinatorial auction generalizations," *Adaptive Agents and Multi-Agent Systems*, vol. 1, no. 1, pp. 69–76, 2002, doi: 10.1145/544741.544760.

[45] J. Fax and R. Murray, "Information flow and cooperative control of vehicle formations," in *IEEE Transactions on Automatic Control*, no. 9, 2004, vol. 49, pp. 1465–1476, doi: 10.1109/TAC.2004.834433.

[46] C. Schumacher and P. Chandler, "Task allocation for wide area search munitions," in *Proceedings of the American Control Conference*, no. 1, 2002, vol. 3, pp. 1917–1922, doi: 10.1109/ACC.2002.1023915.

[47] D. Davis, T. Chung, M. Clement, and M. Day, "Consensus-based data sharing for large-scale aerial swarm coordination in lossy communications environments," presented at 2016 International Conference on Intelligent Robots and Systems, Daejeon, Korea, 2016.

[48] W. Ren, R. Beard, and D. Kingston, "Multi-agent kalman consensus with relative uncertainty," in *Proceedings of the American Control Conference*, no. 1, 2005, vol. 3, pp. 1865–1870.

[49] E. Frew and T. Brown, "Networking issues for small unmanned aircraft systems," *Journal of Intelligent and Robotic Systems*, vol. 54, no. 1–3, pp. 21–37, 2009, doi: 10.1007/s10846-008-9253-2.

[50] H. Choset, "Coverage of known spaces: The boustrophedon cellular decomposition," in *Autonomous Robots*, no. 3, 2007, vol. 9, pp. 247–253, doi: 10.1023/A:1008958800904.

[51] K. Giles, "Mission-based architecture for swarm composability," Ph.D. dissertation, Graduate School of Engineering and Applied Sciences, Naval Postgraduate School, Monterey, CA, 2018.

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