# Applying Monte Carlo Tree Search to the Persistent Surveillance Problem

Patrick Washington
*Aeronautics and Astronautics*
*Stanford University*

***Abstract*—This project develops an online decision maker for the persistent surveillance problem by applying Monte Carlo Tree Search. The problem includes stochasticity in battery usage. The solution incorporates parameter learning to estimate the probability of experiencing a disturbance that affects battery life. The implementation builds off of previous work that used a greedy, deterministic approach in an attempt to account for uncertainty in battery usage while maintaining computation time that is reasonable for online execution. MCTS and a modified version were tested in simulation.**

## I. Introduction

Quadrotors can be very useful in many tasks but have one major limitation: battery life. When discussing their capabilities, it is important to keep in mind that they simply can not fly for very long periods without charging or replacing their batteries. A motivating example is persistent surveillance, where a quadrotor may be able track targets or monitor a region very well but not for an extended period. In order to maintain constant coverage for periods on the order of days, weeks, or months, there must be some autonomous method for deciding where to send the agents and when the transfers should happen. It is not feasible to have people standing by at all times to decide when swaps are necessary even if humans were capable of calculating the best actions to take in a reasonable amount of time.

## II. Background

### A. Prior Work

Prior to this class, I worked on a greedy solution method for a deterministic version of the persistent surveillance problem. The method uses the Hungarian method [1] to find the optimal perfect matching between a set of agents and a set of tasks by assigning costs to each possible assignment. Costs are generated every time step and can be based on battery levels, distances, or rules put in place by the designer. This method can run very quickly, with solutions taking small fractions of a second for even very large systems (tests up to 100 agents). The method can accommodate tasks that might suddenly change by simply changing the cost functions. It can also account for uncertainty by modifying the cost functions.

The main drawback is that it does not look ahead in time very well as it solves for the best matching using only the immediate costs. Cost functions can be designed to try to look ahead but this does not always help. A common occurrence is an agent waiting to take an action without considering that the cost to perform that action will increase at subsequent time steps. This is where MCTS has room to improve the decision making process. There are few realistic systems where decisions have to be made every tenth of a second, meaning that there is time to explore the action space more thoroughly. A couple example simulations of this algorithm are included in Section VIII. One performs well but still has non-ideal transfers. The other is too greedy, demonstrating a system that is not sustainable.

For this project, this algorithm was modified to consider a stochastic system and fit in with the requirements for MCTS.

### B. Related Work

The task assignment problem is a combinatorial optimization problem that can grow quite quickly with the number of agents and tasks. Several methods exist for solving task assignments, including the Hungarian method [1] and Auction algorithms [2], which minimize costs (or maximize rewards) for a single assignment. Solving optimally for several sequential assignments proves difficult computationally for all but the simplest systems because the branching factor leads to many possible assignments. One attempt to approximately solve the multi-step problem is [3]. It tests a subset of possible sequences to reduce the computational complexity.

There has been some work that approaches the persistent surveillance problem from an optimal control perspective as well [4,5,6]. These use dynamic programming to solve for a policy that is proactive toward possible disturbances. The results are policies that anticipate disturbances in battery usage and random failures by sending replacements early or having redundant agents on important tasks. The downside is the balance between computation time and system complexity. These papers use discrete positions that are distances from the base. The policy in [4] deals with a very small state space. There are only 3 agents, 3 possible positions, and 16 possible battery states, yet the policy took 36 hours to compute with exact dynamic programming. The computation speed was improved in [5] by using approximate dynamic programming that can adjust the policy as the model might change during a test but there is still an issue with dimensionality when using offline methods. [6] extends the problem but changes the solution strategy to accommodate larger systems that would be essentially impossible to solve using dynamic programming. The problem is decomposed into several MDPs with a hierarchy to help limit the required computation time, scaling with

the number of agents in the components rather than the total number of agents.

Monte Carlo Tree Search has proven successful at playing some games. Famously, AlphaGo [7] beat a professional Go player in 2015, the first computer player to do so. Go did demonstrate an issue, which is that MCTS may not find branches that lead to wins or losses if there are very few of them. MCTS is also used in the 2014 game Total War: Rome II [8], where the AI decides its next moves with MCTS.

## III. Model

The state space and action space of the system is a combination of the possible states and actions for each individual agent. The state-action space of one agent is independent of the other agents.

### A. State

The state of an agent consists of the position and battery level. The position is a two-dimensional vector containing the $x$ and $y$ displacement from the charging station. Any position that is not at the origin is flying. Denote the position of agent $i$ at time $t$ as $\boldsymbol{p}_t^i$. The battery level is a timer that gives the flight time remaining assuming a nominal discharge rate with no disturbances. Denote the battery level of agent $i$ at time $t$ as $b_t^i$. The state $\boldsymbol{s}_t^i$ of the agent is a vector that concatenates the position and battery level.

$$\boldsymbol{s}_t^i = \begin{bmatrix} \boldsymbol{p}_t^i \\ b_t^i \end{bmatrix} = \begin{bmatrix} x_t^i \\ y_t^i \\ b_t^i \end{bmatrix} \tag{1}$$

The state of the system with $N$ agents at time $t$ is a concatenation of the individual agent states.

$$\boldsymbol{s}_t = \begin{bmatrix} \boldsymbol{s}_t^1 \\ \boldsymbol{s}_t^2 \\ \vdots \\ \boldsymbol{s}_t^N \end{bmatrix} \tag{2}$$

### B. Action

The action at a time step is the assignment for each agent. The assignments can be to charge, to replace a task, or to stay on a task.

$$\boldsymbol{a}_t = \begin{bmatrix} a_t^1 \\ a_t^2 \\ \vdots \\ a_t^N \end{bmatrix}$$

The action space has some restrictions

- An agent that is currently charging may stay on the charger or replace a task.
- An agent that is on the way to replace a task must not change goals until it has reached the task.
- An agent that is on a task may stay on the current task or return to a charger.
- If a replacement arrives at a task, the agent performing that task must leave the task.
- Agents may leave tasks without replacement if needed in the case of low battery.

The replacement tasks are mainly for bookkeeping convenience. These assignments are equivalent to the tasks but kept separate to more easily maintain one robot per task and should have no impact on the solution. The other restrictions act to reduce the action space by removing actions that are definitely inferior to others. Without these restrictions, agents could be directed to any station at any time. This would force the solver to consider actions that send many agents to one task or have agents switch directions every time step. It can be shown using the triangle inequality and intermediate value theorem that changing assignments partway to a station wastes flight time, and therefore battery charge, so forcing the agents to reach a station before switching is not restrictive in terms of performance. There may be edge cases where these restrictions leave out the best action. However, this possibility is not as impactful as reducing the size of the action space by orders of magnitude to make the solution computationally feasible.

### C. Transition Model

Since all agents independently follow the same transition model, consider the transitions for a single agent over a time step of $\Delta t$. Assume that $\Delta t$ is 1 since all distances and battery life can be scaled to match this.

The position is deterministic. If there is a disturbance, the agent uses more energy to maintain the desired trajectory. The agent is given a goal by the action and moves toward the goal at speed $v$ during each time step. Assume $v = 1$ with scaling similar to $\Delta t$. For goal $\boldsymbol{g}$

$$\boldsymbol{p}_{t+1}^i = \begin{cases} \boldsymbol{p}_t^i + \frac{\boldsymbol{g} - \boldsymbol{p}_t^i}{||\boldsymbol{g} - \boldsymbol{p}_t^i||} & \text{if } ||\boldsymbol{g} - \boldsymbol{p}_t^i|| > 1 \\ \boldsymbol{g} & \text{otherwise} \end{cases} \tag{3}$$

The battery transitions are stochastic as the agent maintains its desired trajectory even when there is a disturbance such as a gust of wind. For this problem, a disturbance during a time step doubles the nominal battery discharge. Additionally, assume the charge and nominal discharge rates are both equal to 1 time step of nominal flight unit per time step. Similar to other constants, time and maximum battery levels can be scaled to match this value. Using a probability of disturbance $P(\text{gust})$ and maximum battery level $b_{\max}$, the transition of the battery level is given by

$$b_{t+1}^i = \begin{cases} \min\{b_t^i + 1, b_{\max}\} & \text{charging} \\ \max\{b_t^i - 1, 0\} & \text{flying, } P = 1 - P(\text{gust}) \\ \max\{b_t^i - 2, 0\} & \text{flying, } P = P(\text{gust}) \end{cases} \tag{4}$$

It may be unrealistic to assume equal charge and discharge rates as quadrotors discharge faster than they charge. The ratio of discharge rate to charge rate mainly serves to increase the required number of agents, as will be discussed in Section IV, which increases the computational complexity with minimal effect on the insight a solution to the problem might provide.

### D. Reward Model

The reward model acts to impose the following logic on the system. First, it is bad to waste flight time, so penalize flight time. It is worse to leave required stations unattended, so penalize empty stations with higher weights than those for flight time. Finally, an agent should leave its station if the alternative is dying, so weight dead agents the highest of all three.

$$|W_{\text{fly}}| \ll |W_{\text{empty}}| \ll |W_{\text{dead}}| \tag{5}$$

At a given time step, the reward/penalty is calculated by counting the number of flying agents, the number of empty task stations, and the number of dead agents.

$$r = -W_{\text{fly}} N_{\text{fly}} - W_{\text{empty}} N_{\text{empty}} - W_{\text{dead}} N_{\text{dead}} \tag{6}$$

A minor modification involves only counting the number of flying agents that exceeds the number required for tasks. Practically, this makes no difference as it raises all well-performing systems equally. It does have an effect on systems that leave task stations empty, but the penalty for that should be sufficiently high to negate any meaningful effect.

## IV. Team Sizing

Before introducing the solution methods, it is worthwhile to briefly discuss how the number of agents on the team impacts the performance of the system as it provides justification for the rewards and what should be considered optimal. For the moment, consider a very simple system of $N$ agents and $N$ chargers where the task is to hover at some finite altitude $h$ directly above the charger. At any given time, there must be at least $M$ agents hovering at altitude $h$. This means that before an agent can descend to the charger, another agent must ascend to $h$. The batteries charge and discharge deterministically at rates $r_c$ and $r_d$ per time step, respectively. Altitude changes by 1 unit per time step when commanded.

This system removes any requirement for an intelligent decision maker since when an agent needs to return, the charging agent with the highest battery level ascends. Thus, the only factor on performance for given paremeters is the team size $N$. Note that dead agents (i.e. $b^i = 0$) can be considered to decrease $N$ rather than being considered separately.

A simple performance metric that has shown itself to be useful is the sum of all individual agent batteries, denoted $B$. From time step to time step, $B$ changes by

$$\begin{aligned}
\Delta B &= N_{\text{charging}} r_c - N_{\text{flying}} r_d \\
&= (N - N_{\text{flying}} - N_{\text{full}})\, r_c - N_{\text{flying}} r_d \\
&= \left[ N - N_{\text{flying}} \left(1 + \frac{r_d}{r_c}\right) - N_{\text{full}} \right] r_c \\
&= \left[ N - (M + N_{\text{moving}}) \left(1 + \frac{r_d}{r_c}\right) - N_{\text{full}} \right] r_c
\end{aligned} \tag{7}$$

This is not in a particularly useful form. Consider the idea that for every discharging agent, there must be $r_d/r_c$ charging agents to balance the $B$ gains and losses. Since there must be at least $M$ discharging agents at a given time, a lower bound on $N$ is $M(1 + r_d/r_c)$ to maintain a balanced $B$. Introduce a term $N_{\text{extra}}$ that represents the difference between $N$ and the lower bound. This is the design parameter for the team size. Note that unlike the other counts, $N_{\text{extra}}$ does not need to be an integer if $r_d$, $r_c$, and $M$ lead to a non-integer. Continue Eq (7) using the above

$$\begin{aligned}
\Delta B &= \left[ N - (M + N_{\text{moving}}) \left(1 + \frac{r_d}{r_c}\right) - N_{\text{full}} \right] r_c \\
&= \left[ N_{\text{extra}} - N_{\text{moving}} \left(1 + \frac{r_d}{r_c}\right) - N_{\text{full}} \right] r_c
\end{aligned} \tag{8}$$

This final form gives insight into the team sizing and decision making. First, it is clear that $N_{\text{extra}}$ must be positive because otherwise $\Delta B$ can never recover from losses. It also shows diminishing returns on increasing $N_{\text{extra}}$ because every agent with a full battery cancels out one of the extra agents. Finally, it shows that agents that are moving between stations, in this case the ground and the desired altitude, have the strongest negative impact on $B$. Having many full batteries may be wasteful but sending an agent into the air just because it is full would be a bad decision. Plots showing how the total battery evolves in cases with and without enough agents are shown in Fig. 1 and 2. The animations of the simulations that produced these plots are included in Section VIII.

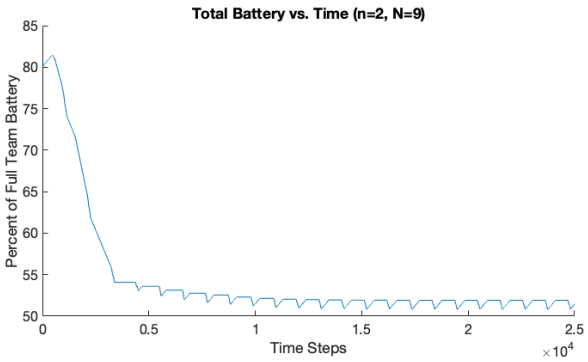


Fig. 1. Total battery versus time in a test with enough agents to be sustainable ($N_{\text{extra}} = 1$). After an initial settling period, the system reaches steady state oscillations.

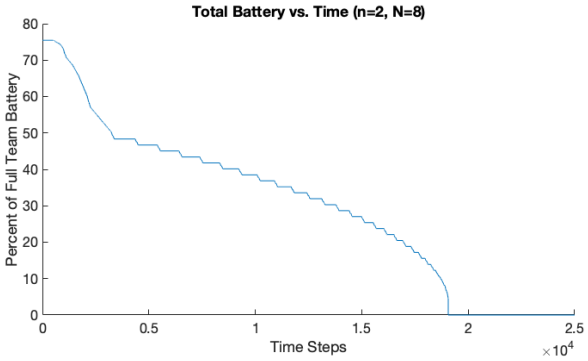


Fig. 2. Total battery versus time in a test without enough agents to be sustainable ($N_{\text{extra}} = 0$). The system has no way to increase its battery level.

Of course, the problem can be made easier by having many more agents than required but this is not realistic when it comes to implementing a real system. Quadrotors are not free and having many extra is wasteful.

The above demonstrates the main goal of the decision maker in the surveillance problem. The decisions should work to maintain surveillance coverage and prevent agents from dying while minimizing the amount of time spent moving between stations. The optimal policy will leave no stations unattended, keep all agents alive, and have the minimum amount of flight time.

It is important to distinguish between the number of transfers and the total time spent transferring from station to station. Many short transfers could very well be better than fewer long transfers when considering the total battery capacity of the team.

## V. Monte Carlo Tree Search

The state space for this problem is very large. Consider a system with 100 battery levels and 6 agents. There are $10^{12}$ states without even considering the agents' positions. However, from any individual system state, there are relatively few states that the system can reach in one time step. The positions change deterministically and each battery can only transition to 1 or 2 levels depending on whether the agent is charging or flying. Additionally, with the restrictions on the action space described previously, the number of possible actions from any given state is much smaller than the space of all actions. This means that the problem lends itself to using online methods.

Monte Carlo Tree Search is an anytime online algorithm [9]. It searches future states by sampling, returning an action after a set amount of time. As the search progresses, the values at station-action pairs are updated to provide an estimate for how well an action or series of actions will perform. The search balances exploration and exploitation to simulate actions based on an estimate for the upper bound of performance.

### A. Search

When searching the action space, the system chooses the action with the highest value of

$$Q(s,a) + c\sqrt{\frac{\log \sum_a N(s,a)}{N(s,a)}} \tag{9}$$

where $Q(s,a)$ is the estimated value of taking action $a$ at state $s$, $N(s,a)$ is the number of times $(s,a)$ has been visited, and $c$ is a parameter that controls how much to value exploration versus exploitation. This balances exploiting the best action found (highest $Q(s,a)$) and exploring unvisited regions (low $N(s,a)$). When a new value $q$ resulting from a simulation is returned, the relevant counter increments and the estimate of the value function updates with

$$Q(s,a) \leftarrow Q(s,a) + \frac{q - Q(s,a)}{N(s,a)} \tag{10}$$

This update method proved to be quite troublesome so the final system used a modified update method.

### B. Modification

In this problem, most of the actions from a given state result in poor performance. Unfortunately, this means that the update method that averages the discounted rewards can have a negative influence on the few good nodes that exist. Since most of the children from a given node are bad, their scores can easily overwhelm the scores from the good paths down the tree. Using discount factors helped to some degree but the problem remained. An alternative to the averaging method that proved helpful was to use the maximum value seen instead of an expected value while also increasing the exploration parameter. The search still uses Eq (9) to traverse the tree and the rollout is identical but the $Q(s,a)$ values are the best $q$ values seen after taking action $a$ from state $s$. The $q$ value may still be discounted if desired.

$$Q(s,a) \leftarrow \max\{Q(s,a), q\} \tag{11}$$

Increasing the exploration parameter was important because the downside of using the best values seen is that getting trapped in exploitation is very easy after a single good run.

### C. Rollout

The rollout is used to estimate values by executing a default policy to some depth. This system uses the previously developed greedy method to quickly evaluate many time steps into the future. In this case, the costs are based on calls for help and battery level. An agent performing a task needs a replacement when its battery level reaches the point where a replacement agent will reach the task just before the agent needs to leave for the charger. In the rollout, an agent that needs replacing sends out a distress signal to call for the other agents to help. When this happens, the agents that are charging submit costs based on their battery level, favoring those with higher charges. The rollout returns rewards (penalties) based on the previously described reward model, adding a discount factor to weight the score toward earlier states.

### D. Parameter Learning

In addition to making the decisions about where to send the agents, the system estimates the probability of an agent experiencing a disturbance during a time step. The system uses a beta distribution with uniform prior. After each executed step, the system gathers the observations from all of the agents and updates the counts. The system then uses the maximum likelihood estimate over the resulting probability density function as the next probability to use in simulation.

$$\hat{P}(\text{gust}) = \arg\max_P \text{Beta}(P \mid 1 + \text{gusts}, 1 + \text{flight-gusts}) \tag{12}$$

Note that the estimate and counts are not updated during the simulation phase of the search. The simulations use a constant probability when applying the transition model to steps through the tree or the rollout policy.

Another possibility for the estimate could have been an upper confidence bound such as the probability where the cumulative density function is 0.75. This would generally give

a higher estimate for the probability. Using a higher value would make the search slightly more cautious about allowing agents to have lower battery levels since it would expect faster battery drain. In reality, the beta distribution quickly converges to a spike at the true gust probability so any difference caused by the estimation method is negligible.

## VI. Results

The algorithm was implemented in MATLAB. For my tests, I used 6 robots with 2 moving stations. The batteries nominally lasted 500 time steps but the real battery life was somewhat reduced by the gust probability of 0.1. The estimate for the gust probability started at 0.5 but the initialization had very minimal effect beyond the first time step. The search was not allowed to start a new simulation after 2 seconds had passed. I had some tests with much longer computation times on the order of minutes but the results were not different enough to justify the infeasible computation time required for thousands of executed time steps. Additionally, at some point, the algorithm can no longer be reasonably considered as a feasible online method as far as actual implementation is concerned. Animations from sample simulations can be found in Section VIII.

As mentioned in the previous section, the base version of MCTS performed poorly. Since most actions lead to agents moving stations, the search tended to result in an action that results in a transfer. Upon inspecting the values after every calculation, the following pattern emerged in cases where the agents performing tasks did not need replacing. After one pass, the solver would tend to like the one action that did not send any replacements. As such, it moved its search toward that action. Unfortunately, there is only one action that does not send any replacements at the next level down. After expanding the descendants of the initially favored action, most of the scores come back very negative, overwhelming the initial positive score. These negatives actually pushed the $Q(s, a)$ value below some other actions that were definitely worse.

As seen in the animations, the base MCTS implementation has far too many transfers, which leads to reduced charging times and dead batteries. Changing parameters had little effect on the results. Even increasing the allowed computation time did not help very much. This result is not wholly unexpected in hindsight. MCTS relies on sampling the space to find the best expected value. If there is only one good path, in this case one that does not waste battery, out of the many possibilities, it is difficult to find without scores being corrupted by nearby bad paths. MCTS also relies on the idea that neighboring states are similar, which is often not true in this problem.

The version of the search with modified value updates performed better but it still had some obvious flaws. There were far fewer transfers, which was a very important difference. However, the transfers did not have very good timing and the choice of agents to send out as replacements was not always ideal. As can be seen in the animation, there were as many transfers over a long distance as there were over shorter distance, leading to the conclusion that the system did not adequately consider travel distance when deciding when to send replacements. Additionally, the replacement agent was frequently not the agent with the highest battery level. I suspect that this could be improved by looking to a longer horizon because when agents with less charge were sent out, they had high enough charge that the effects would not be seen until well past the horizon. Unfortunately, looking further ahead would limit how many samples could be considered before a decision is required.

The performance of the parameter learning was always very good, as shown in Fig. 3. Note that the resolution of the pdf used to esimate the probability was 0.01, so jumps of 0.01 are easily possible from a single time step with or without a disturbance.

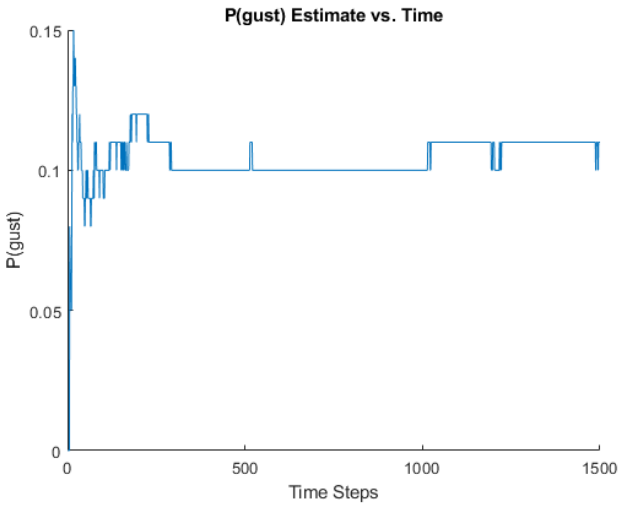


Fig. 3. Estimate of P(gust) over time. The true value is 0.1. Note that the jumps to 0.11 are within margin of error as the resolution of probabilities in the beta pdf was 0.01.

## VII. Future Work

I intend to continue testing MCTS to see if any parameter values help the solution. Something to watch for is whether the parameters are very specific to the exact problem. Since it seems like MCTS may not be well suited to the persistent surveillance problem, it may be beneficial to implement a branch and bound forward search [9], which is the direction that the modified MCTS went toward.

## VIII. Videos

Good greedy policy

`https://youtu.be/XW_jNqhqG8k`

This animation shows the performance of the greedy method. The algorithm is very similar to the rollout policy used for this project. When an agent needs replacing, a red circle appears to indicate a call for help. Notice that several transfers happen at the furthest distance. This is a behavior that should be improved by a less greedy method. The size of the squares and the number beneath the squares indicate the

battery charge level. The positions at the bottom are charging stations.

Bad greedy policy

https://youtu.be/q3jYZgk5oAA

This animation demonstrates a bad example of the system behaving greedily. At every time step, if an agent thinks it is better suited to the task, it takes over. This leads to far too much travel time and an unsustainable system.

Poorly-sized team

https://youtu.be/nn-rcmrSQac

This animation demonstrates the need to account for travel time. The team is sized to exactly balance battery usage when there are no transfers or full batteries. The agents in this system discharge at 3 times the charge rate. The size of the squares and the number beneath the squares indicate the battery charge level.

Well-sized team

https://youtu.be/XqrXy6kv_uw

This animation demonstrates the benefit of a single extra agent. This agent allows the system to recover from the battery loss of a transfer. This system can run indefinitely. The size of the squares and the number beneath the squares indicate the battery charge level.

Original MCTS

https://youtu.be/q_XrXh1sBt4

This animation shows the result of the base MCTS implementation. The system commands too many transfers and ultimately has some agents run out of battery. This is similar to the bad greedy video. The chargers are all grouped into one point in the animation.

Modified MCTS

https://youtu.be/z5eUeK5OGBQ

This animation shows the result of the modified MCTS implementation. The system appears fine over the duration of the test but some decisions are not ideal, likely because issues would not appear for hundreds of time steps later, which is beyond the horizon of the search. The chargers are all grouped into one point.

All code can be found at

https://drive.google.com/drive/folders/1A9prKQAY2J_0i3y9cLR_cgHuk0UzatZ1?usp=sharing

All functions are my own except for munkres.m. The license provided with that function is included in the folder.

## REFERENCES


[1] H. W. Kuhn, "The Hungarian method for the assignment problem," Naval Research Logistics Quarterly, vol. 2, no. 1-2, pp. 83–97, 1955.

[2] D. P. Bertsekas, "Auction Algorithms," Encyclopedia of Optimization, pp. 73–77.

[3] H.-L. Choi, L. Brunet, and J. How, "Consensus-Based Decentralized Auctions for Robust Task Allocation," IEEE Transactions on Robotics, vol. 25, no. 4, pp. 912–926, 2009.

[4] B. Bethke, J. P. How, and J. Vian, "Group Health Management of UAV Teams with Applications to Persistent Surveillance," 2008 American Control Conference, 2008.

[5] B. Bethke, J. P. How, and J. Vian, "Multi-UAV Persistent Surveillance with Communcation Constraints and Health Management," AIAA Guidance, Navigation, and Control Conference, Aug. 2009.

[6] Y. F. Chen, N. K. Ure, G. Chowdhary, J. P. How, and J. Vian, "Planning for Large-Scale Multiagent Problems via Hierarchical Decomposition with Applications to UAV Health Management," 2014 American Control Conference, 2014.

[7] "AlphaGo," DeepMind. [Online]. Available: https://deepmind.com/research/alphago/. [Accessed: 07-Dec-2018].

[8] "Monte-Carlo Tree Search in TOTAL WAR: ROME II's Campaign AI," Top 10 Most Influential AI Games — AiGameDev.com. [Online]. Available: http://aigamedev.com/open/coverage/mcts-rome-ii/. [Accessed: 07-Dec-2018].

[9] Kochenderfer, M. (2015). Decision Making Under Uncertainty. 1st ed. Cambridge, MA: The MIT Press, pp.99-103.