Maximizing Surveillance and Minimizing Risk of Detection for Unmanned Aerial Vehicles

David R Crow, 2d Lt, USAF

***Abstract*—In this research, we attempt to solve a multi-domain optimization problem using various algorithmic techniques. Specifically, we utilize different deterministic, stochastic, and local search algorithms to maximize unmanned aerial vehicle (UAV) surveillance coverage and minimize the risk of detection for each UAV. We formally describe the problem's domain before following the algorithm domain design process to refine each search algorithm from problem domain to implemented code. After doing so, we evaluate the software implementations and present the experimental results obtained for different problem instances. Finally, we discuss variations on these algorithms and other potential techniques for solving this optimization problem.**

***Index Terms*—algorithm design, multi-domain optimization, guided search, software analysis**

Contents

List of Acronyms

2D two-dimensional
ACO ant colony optimization
GA genetic algorithm
IDA* iterative deepening A*
IDE integrated development environment
LBS local beam search
MCP maximum coverage problem
MRP minimum risk problem
NP nondeterministic polynomial time
NP-C nondeterministic polynomial time-complete
PD/AD problem domain/algorithm domain
SA simulated annealing
SBS stochastic beam search
UAV unmanned aerial vehicle
USAF United States Air Force

I. Introduction

Consider the following problem: the United States Air Force (USAF) possesses a set of UAVs, all identical, and each with a limited flight time. The Air Force wants to conduct surveillance over City X in Country Y. City X, however, possesses a set of UAV detectors distributed throughout the city limits. How can the USAF surveil as much of City X as possible without attracting unnecessary attention?

This research concerns the problem of maximizing the coverage of an area with a swarm of UAVs while minimizing the risk of detection to said UAVs. Because both the *maximum coverage problem (MCP)* and the *minimum risk problem (MRP)* are nondeterministic polynomial time (NP)-complete (NP-C) problems, the multi-domain optimization of both MCP and MRP is at least NP-hard [1]–[3]. (We prove the exact complexity class of MCP and of MRP in Appendix A.)

To approximate solutions to example problem instances, then, we utilize several different algorithmic techniques: deterministic search, stochastic search, and local search. We compare the results of each technique to determine which search algorithm best solves the problem at hand. Because optimally solving a problem as difficult as MCP/MRP is impossible (unless $P = NP$ [4]), we cannot guarantee a

TABLE I
THE PROBLEM DOMAIN/ALGORITHM DOMAIN PROCESS FOR EFFECTIVE, EFFICIENT ALGORITHM DESIGN

#	Step
1	Define/analyze the problem domain
2	Choose an algorithm domain specification strategy
3	Evolve a general solution design specification
4	Refine solution design recursively to low-level design
5	Map low-level design to selected programming language
6	Evaluate implementation and document process

perfect solution; however, our results show that approximate solutions are feasible.

This research employs the problem domain/algorithm domain (PD/AD) process for algorithm design to generate all algorithms [5]–[7]. Table I shows the PD/AD process. We follow this process for each of the algorithmic techniques (deterministic, stochastic, and local). Step 1 is the same for all three techniques, so it is explained in Section II. Sections III, IV, and V detail the remaining steps for the deterministic, stochastic, and local approaches, respectively. Section VI describes the testing and evaluation process, for which we use the reporting approach found in [8]. Finally, Section VII discusses the conclusions one can draw from this research.

II. Problem Domain

In English, this research concerns the problem of conducting surveillance in a designated, square-shaped area consisting of $d \times d$ unit squares. Enemy sensors are placed throughout this area; every sensor has the same sensing radius r. The set L contains the coordinate location for each sensor. We have a specific number n of UAVs on hand; every UAV has the same battery capacity b.

The goal is to fly the UAVs through the designated area so as to maximize surveillance capabilities. Each UAV starts and ends its flight outside the designated area (start and finish do not have to be in the same location). The UAV depletes one unit of battery per unit square visited. The current constraints, of course, imply that a UAV cannot sit in a non-edge square with a remaining capacity of zero. Because each unit square can be uniquely observed exactly once, we wish to maximize the total number of unique squares visited by all UAVs.

We also wish to minimize risk. Because the sensors each have a known sensing radius, up to one-hundred percent of each unit square can be watched by the surrounding sensors. If (say) thirty percent of a unit square is covered by some combination of sensors, then each UAV that passes through said square has a thirty percent chance of being detected. We wish to minimize the sum of each UAV's average chance of detection.

Of course, the individual impact of coverage and risk values must be finely-tuned: if coverage is too important, the UAVs will accept too much risk; if coverage is not important enough, the UAVs will refuse to visit squares with any risk.

Fig. 1 displays a specific MCP/MRP instance. One can clearly see the constraints on the system and the desired outcome by referencing this figure. Because MCP/MRP is such a difficult problem, this simple problem instance already

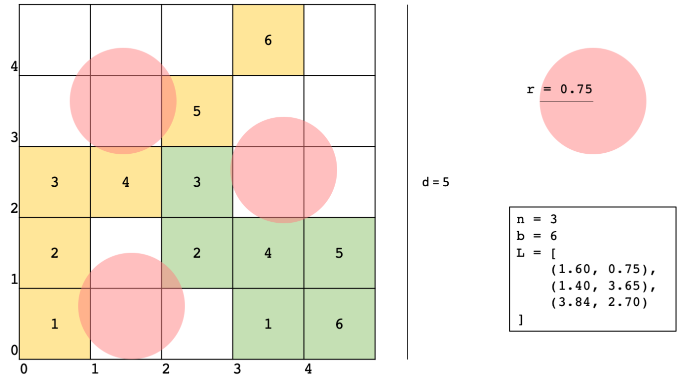

Fig. 1. An example problem instance. The green and yellow squares depict the paths for two UAVs. The red circles depict the coverage area for each of three UAV detectors.

entails a vast search space [9]–[11]. Thus, our algorithmic techniques must filter out poor solutions and exploit patterns in the space to identify adequate solutions.

Mathematically, the problem domain for MCP/MRP can be summarized as follows:

- D_i, input domain
  - n, the number of UAVs available for scheduling
  - b, the (integer) battery capacity of each UAV
  - L, a list of all enemy sensor (x, y) locations
  - r, the sensing radius of the enemy sensors
  - d, the (integer) side length of the surveillance area
- Input conditions
  - $\forall i \in \{n, b, |L|, r, d\}, i > 0$; each of n, b, $|L|$, r, d must be greater than 1 to ensure the problem is nontrivial
  - $\forall (x, y) \in L, 0 \leq x < d, 0 \leq y < d$; each sensor must be within the bounds of the surveillance area
- D_o, output domain
  - P, the set of UAV paths
  - $V_P = 0.90 \times total\_ratio + 0.10 \times edge\_ratio - 0.25 \times (same\_repeats + other\_repeats) - 0.50 \times risk$
    - $total\_ratio = squares\_covered \div d^2$
    - $edge\_ratio = edges\_covered \div (4d - 4)$
    - $same\_repeats$, number of times a UAV visits a square it has already visited
    - $other\_repeats$, number of times a UAV visits a square another UAV has already visited
    - $risk = total\_risk \div path\_length$
- Output conditions
  - $|P| = n$; every UAV must have a path (even if the path is empty)
  - $\forall p \in P, |p| \leq b$; no UAV can fly more than b unit squares
  - $\forall p \in P,\ edge(first(p)) = 1,\ edge(last(p)) = 1$; every UAV must start and end on an edge square
  - $\forall Q,\ Q \neq P \Rightarrow V_Q < V_P$; the value of the path set P must be maximal

TABLE II
THE STANDARD ELEMENTS IN SEARCH ALGORITHM DESIGN

Element	Description
Set of candidates	A set of possible next-state candidates (implicit or explicit)
Next-state generator	Generates the next-state candidates from a parent state candidate
Feasibility	Determines if various next-state candidates meet the feasibility criteria or constraint
Selection	A function to select/extract/delete one or more of the feasible next state candidates
Solution	A function to determine if the current-state candidate is an acceptable solution
Objective	A function that reflects the selected optimization criteria
Heuristics	A function that, as appropriate, contains algorithmic strategies usually based upon a reduced set-of-candidates via insight from the problem domain's structure and the objective function

- Objective
  - Maximize V, the overall state value
    - Maximize $\sum_{i=1}^{n} squares\_covered_i$
    - Minimize $\sum_{i=1}^{n} risk_i$
  - Employ deterministic, stochastic, and local algorithmic approaches to identify the best possible solution
  - Conduct performance analyses of the approaches
- Problem domain complexity
  - MCP is NP-C
  - MRP is NP-C
  - See Appendix A for more information (including NP-C proofs and a discussion of the overall MCP/MRP complexity)

III. DETERMINISTIC SEARCH

According to [12], "a deterministic algorithm is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states." In our first attempt at solving MCP/MRP, we employ a well-known determistic search algorithm: A* [13]. A* is one of the premier search algorithms in computer science because it is easy to understand and to implement. Additionally, when the user's heuristic is admissible (for a tree search) or consistent (for a graph search), A* is guaranteed to return the optimal solution [14]–[16]. This is a positive for the USAF, of course, because it means that A* will always give the correct solution to MCP/MRP – assuming we can devise a suitable heuristic.

Of course, we must correctly utilize and document the PD/AD process to develop an effective algorithm. Specifically, we must define and refine the standard search constructs (see Table II) from the algorithm domain (see Section II) to our exact program implementation [5].

The remainder of this section discusses Steps 2–6 for the deterministic approach to MCP/MRP. Details for these steps can be found in Sections III-A, III-B, III-C, and III-D.

A. Design Specification

Step 2 requires that we select an algorithm domain specification strategy. In essence, this step defines the standard search constructs from a high vantage point. We do this as follows:

- Set of candidates
  - A* stores its set of candidates in an `open` list
  - To reduce the search space, we also store *visited* candidates in a `closed` list
- Next-state generator
  - Given an empty world of size $d \times d$, we create $P(4d-4, n)$ initial next-states, one for each starting configuration of n UAVs
  - For a given world, the set of next-states includes all copies of the world with one additional movement of one UAV
    - *Constraint*: the UAV's path length p must be less than b
    - *Constraint*: the UAV cannot move to a non-edge square if $p = b - 1$
- Feasibility
  - For a given state S, determine whether, for every UAV $u \in S$, u is on an edge square and either *a)* remaining_battery(u) $= 0$ or *b)* every square $s \in S$ is covered
  - Determine whether S meets constraints defined in the problem domain
- Selection
  - A* always selects the candidate with the lowest $f(S) = g(S) + h(S)$, where S is the state, $g(S)$ is the cost to reach S from the start, and $h(S)$ is the estimate of the remaining cost to the goal
  - Delete those candidates for which $\exists u \in$ UAVs such that remaining_battery(u) $= 0$ and u is not on an edge square
- Solution
  - If *feasibility* constraints are met and V_S is minimized, return S
- Objective
  - Minimize $f(S)$ using *heuristics* (defined in Section III-B) while meeting the *feasibility* constraints

B. Design Refinement

In Steps 3 and 4 of the PD/AD process, one must evolve a general solution design specification and instantiate the problem design. This section includes a discussion of the required abstract data types and a refinement of the standard search constructs detailed in Section III-A.

Because the end-goal is functional Python code, we must specify the data structures used to solve MCP/MRP. As stated previously, A* requires `open` and `closed` sets, so we use simple Python lists to store all new and previously-visited states. For this research, lists are effecient and effective, and we feel no need to reinvent the wheel. At this somewhat-abstract level, there does not seem to be a need for other complex data structures – only primitive data types.

We now refine our design specification so as to bridge the gap between the theoretical algorithm domain and the pseudocode. The refined standard search constructs are as follows:

- Set of candidates
  - open $= \emptyset$
  - closed $= \emptyset$
- Next-state generator
  - If $S = \emptyset$, $\forall c \in$ start_configurations, $S = S \cup \{c\}$
  - Otherwise, $\forall u \in$ UAVs and $\forall a \in$ adjacent_squares$_u$, $S = S \cup \{u \cup \{u_a\}\}$
    - *Constraint*: $|u| \leq b$
    - *Constraint*: If $p = b - 1$, let adjacent_squares$_u$ = adjacent_edge_squares$_u$
- Feasibility
  - Feasible if, $\forall u \in$ UAVs, edge(location(u)) $= 1$ and one of the following holds:
    - remaining_battery(u) $= 0$
    - $\forall s \in S$, covered(s) $= 1$
  - Additionally, each of these problem domain constraints must hold:
    - $|P| = n$; every UAV must have a path (even if the path is empty)
    - $\forall p \in P, |p| \leq b$; no UAV can fly more than b unit squares
    - $\forall p \in P,\ edge(first(p)) = 1,\ edge(last(p)) = 1$; every UAV must start and end on an edge square
    - $\forall Q,\ Q \neq P \Rightarrow V_Q < V_P$; the value of the path set P must be maximal
- Selection
  - At each time t, select $\arg\min_S f(S)$
  - Delete S if $\exists u \in$ UAVs s.t. remaining_battery(u) $= 0$ and edge(location(u)) $\neq 1$
- Solution
  - Return
    $$\begin{cases} S & \text{if feasibility}(S) = 1 \text{ and } V_s \text{ is maximized} \\ \emptyset & \text{otherwise} \end{cases}$$
- Objective
  - Maximize $f(S)$ using *heuristics* while meeting the *feasibility* constraints
  - As defined in the problem domain, $f(S) = v$, the overall state value, which concerns the following:
    - Maximized $\sum_{i=1}^{n} squares\_covered_i$
    - Minimized $\sum_{i=1}^{n} risk_i$
- Heuristics
  - For a given state S, $h(S)$ concerns the following:
    - $total\_ratio = squares\_covered \div d^2$
    - $edge\_ratio = edges\_covered \div (4d - 4)$
    - $same\_repeats$, number of times a UAV visits a square it has already visited
    - $other\_repeats$, number of times a UAV visits a square another UAV has already visited
    - $risk = total\_risk \div path\_length$
  - $g(S) = \sum_u b -$ remaining_battery$_u$

Note that the heuristic is *not* monotonic. However, careful tuning of the weights over multiple iterations generated this heuristic. All other heuristic weights tested performed poorly when compared to this one. Additionally, although monotonicity is desirable, we note that the size of the search space for MCP/MRP guarantees that a deterministic approach is likely infeasible for large problem instances. See Sections VI and VII for further discussion.

C. Pseudocode

Without an intermediate step between the algorithm design and the actual implementation, one can easily fail to correctly implement one's solution. Pseudocode, the PD/AD process's Step 5 requirement, serves as this intermediary. Because a large portion of our implemented code is used for overhead and for evaluation, the pseudocode given here describes only the essential search function. In other words, we do not give pseudocode for extraneous implementation details (e.g., library imports, output functions, initialization steps); these details can be found in Appendix C.

A* requires a `frontier` (`open`) set and an `explored` (`closed`) set. In our actual implementation, we use Python lists for these sets. Each `world` instance (that is, the two-dimensional (2D) $d \times d$ grid that contains the sensors and the UAV paths) is contained within a 2D Python list; the path sets are also stored as 2D lists. All other data structures are one of the primitive Python types: `string`, `integer`, `float`, or `boolean` [17].

Pseudocode for the deterministic approach – with the standard search elements labeled with comments – is shown in Algorithm 1.

D. Implementation

Step 6 of the PD/AD process requires a mapping from Step 5's pseudocode to an actual, executable implementation. We decided to implement the deterministic approach in Python because it an easy-to-write language, it's moderately-fast, and it has a large, well-documented set of tools. PyCharm Professional 2019.1 served as the integrated development environment (IDE) [18]. All code was written on a 2017 MacBook Pro [19]. Because we do not test exceptionally difficult MCP/MRP instances in Section VI (because MCP/MRP is far too difficult to do so), neither the computer nor the IDE failed to perform.

The code was written using good software development principles. We draw upon our undergraduate- and graduate-level computer science courses in software engineering and upon references [20]–[22] for all questions concerning effective software engineering principles.

We can analyze the complexity of the A* search algorithm by referring to Algorithm 1. The `for-loop` on line 4 performs $P(4d-4, n)$ iterations because there are $P(4d-4, n)$ ways to initialize n UAVs on the edges of a $d \times d$ grid [23]. The inner loop on line 7 iterates over n UAVs, so the loop on line 4 is bounded by $O(n \times P(4d-4, n))$. Because there are $n \times P(d^2, b)$ possible paths for the n UAVs, and because each location in the world has up to eight adjacent squares, the `while-loop` on line 14 is bounded by $O(8n^2 \times P(d^2, b))$.

Algorithm 1 A Deterministic Approach for MCP/MRP

Input: w, an empty $d \times d$ grid
Input: p, a set of n empty paths
Output: p', a set of n paths
Output: v, the value of p' in w

```
// set of candidates
frontier = ∅
explored = ∅
for each starting configuration s of n UAVs in w do
    new_world = w
    new_paths = p
    for all u ∈ UAVs do
        new_world[location(u)] = 1
        new_paths = new_paths_u ∪ {location(u)}
    end for
    v = value(new_world, new_paths)
    frontier = frontier ∪ {v, new_world, new_paths}
end for
while |frontier| ≠ 0 do
    // heuristics
    frontier = frontier.sort() {sort on v}
    // selection
    v, w, p = frontier.pop_front()
    explored = frontier ∪ {w}
    // objective
    if ∀u ∈ UAVs, edge(location(u)) = 1 and every square
    is covered or every UAV's battery is depleted then
        // solution
        return (w, p)
    end if
    // feasibility
    for all u ∈ UAVs do
        if u's battery has 1 unit remaining then
            neighbors = adjacent_edges(u)
        else
            neighbors = adjacent(u)
        end if
        for all q ∈ neighbors do
            w' = w
            p' = p
            w'[location(q)] = 1
            p'_u = p'_u ∪ location(q)
            state = {value(w', p'), w', p'}
            if state ∉ frontier and state ∉ explored then
                // next-state generator
                frontier = frontier ∪ {state}
            end if
        end for
    end for
end while
return "No solution found!"
```

Overall, then, the A* algorithm *for MCP/MRP* has a complexity of $O(n \times P(4d-4, n) + 8n^2 \times P(d^2, b)) = O(n^2 \times P(d^2, b))$. Clearly, this is an exceedingly difficult problem.

The reader can view the full software implementation in Appendix C.

IV. Stochastic Search

Stochastic algorithms are those that generate and use random variables to solve, approximate, or optimize various problems. Examples include simulated annealing, random-restart hill-climbing, and tabu search, among many, many more. These algorithms use random variables to avoid plateaus or locally-but-not-globally optimal solutions [24], [25]. As described in Section VI, the sheer size of the MCP/MRP search space means that our deterministic search algorithm – A* – cannot solve anything but very small problem instances. For this reason, we now attempt to solve MCP/MRP using a stochastic algorithm: stochastic beam search (SBS) [26]. Although SBS is not guaranteed to return an optimal solution, it is all-but-guaranteed to terminate, and often with a reasonable – if not optimal – solution.

In each iteration, SBS generates all unseen neighbors of all `open` nodes. It then probabilistically selects the best k of these neighbors; all others are moved to the `closed` set. (In this research, we let $k = 5$ to ensure termination.) Each node is weighted by its value v; a random number generator (akin to a roulette wheel) selects k nodes, but those with greater v values are more likely to be selected.

As in Section III, we must employ the PD/AD process to develop an effective SBS algorithm. We again refine the standard search constructs (see Table II) from the algorithm domain (see Section II) to our exact SBS implementation.

The remainder of this section discusses Steps 2–6 for the stochastic approach to MCP/MRP. Details for these steps can be found in Sections IV-A, IV-B, IV-C, and IV-D.

A. Design Specification

Step 2 of the PD/AD process requires that we select an algorithm domain specification strategy. We define the standard search constructs for our eventual SBS implementation as follows:

- Set of candidates
  - SBS stores its set of candidates in an `open` list
  - To reduce the search space, we also store *visited* candidates in a `closed` list
- Next-state generator
  - Given an empty world of size $d \times d$, we create $P(4d-4, n)$ initial next-states, one for each starting configuration of n UAVs
  - For a given world, the set of next-states includes all copies of the world with one additional movement of one UAV
    - *Constraint*: the UAV's path length p must be less than b
    - *Constraint*: the UAV cannot move to a non-edge square if $p = b - 1$

- Feasibility
  - For a given state S, determine whether, for every UAV $u \in S$, u is on an edge square and either *a)* remaining_battery(u) $= 0$ or *b)* every square $s \in S$ is covered
  - Determine whether S meets constraints defined in the problem domain
- Selection
  - SBS *attempts* to select the k candidates with the lowest $h(S)$ values, where S is the state, and $h(S)$ is the estimate of the remaining cost to the goal
  - Delete those candidates for which $\exists u \in$ UAVs such that remaining_battery(u) $= 0$ and u is not on an edge square
  - Additionally, probabilistically delete all but the best k candidates
    * To ensure our SBS MCP/MRP solution is feasible, let $k = 5$
    * Random number generation ensures we can't always select the states we want to select – the random number must be less than or equal to $V_S \div \max V_S'$ to select S
- Solution
  - If *feasibility* constraints are met and V_S is minimized, return S
- Objective
  - Minimize $f(S)$ using *heuristics* (defined in Section IV-B) while meeting the *feasibility* constraints

B. Design Refinement

The PD/AD process's third and fourth steps require that one evolves a general solution design specification and instantiates the problem design. In this section, we discuss the abstract data types and structures required in our SBS implementation. Additionally, we refine the standard search constructs previously detailed in Section IV-A.

Step 6 of the PD/AD is functional Python code, so we must first specify the SBS data structures used to solve MCP/MRP. As stated previously, SBS requires `open` and `closed` sets, so, like in our A* implementation, we utilize Python-style lists to store all new and previously-visited states.

As in Section III-B, we must now refine the high-level standard search constructs. This will allow us to map our problem and algorithm domains onto pseudocode and executable code.

- Set of candidates
  - open $= \emptyset$
  - closed $= \emptyset$
- Next-state generator
  - If $S = \emptyset$, $\forall c \in$ start_configurations, $S = S \cup \{c\}$
  - Otherwise, $\forall u \in$ UAVs and $\forall a \in$ adjacent_squares$_u$, $S = S \cup \{u \cup \{u_a\}\}$
    * *Constraint*: $|u| \leq b$
    * *Constraint*: If $p = b - 1$, let adjacent_squares$_u$ = adjacent_edge_squares$_u$
- Feasibility
  - Feasible if, $\forall u \in$ UAVs, edge(location(u)) $= 1$ and one of the following holds:
    * remaining_battery(u) $= 0$
    * $\forall s \in S$, covered(s) $= 1$
  - Additionally, each of these problem domain constraints must hold:
    * $|P| = n$; every UAV must have a path (even if the path is empty)
    * $\forall p \in P, |p| \leq b$; no UAV can fly more than b unit squares
    * $\forall p \in P$, $edge(first(p)) = 1$, $edge(last(p)) = 1$; every UAV must start and end on an edge square
    * $\forall Q,\ Q \neq P \Rightarrow V_Q < V_P$; the value of the path set P must be maximal
- Selection
  - At each time t, probabilistically select up to $k = 5$ $\arg\max_S h(S)$
  - Delete S if $\exists u \in$ UAVs s.t. remaining_battery(u) $= 0$ and edge(location(u)) $\neq 1$
  - Delete S if random_number $> V_S \div \max V_S'$
  - Delete S if |frontier| $\geq k = 5$
- Solution
  - Return

$$\begin{cases} S & \text{if feasibility}(S) = 1 \text{ and } V_s \text{ is maximized} \\ \emptyset & \text{otherwise} \end{cases}$$

- Objective
  - Maximize $f(S)$ using *heuristics* while meeting the *feasibility* constraints
  - As defined in the problem domain, $f(S) = v$, the overall state value, which concerns the following:
    * Maximized $\sum_{i=1}^{n} squares\_covered_i$
    * Minimized $\sum_{i=1}^{n} risk_i$
- Heuristics
  - For a given state S, $h(S)$ concerns the following:
    * $total\_ratio = squares\_covered \div d^2$
    * $edge\_ratio = edges\_covered \div (4d - 4)$
    * $same\_repeats$, number of times a UAV visits a square it has already visited
    * $other\_repeats$, number of times a UAV visits a square another UAV has already visited
    * $risk = total\_risk \div path\_length$

C. Pseudocode

The PD/AD process's Step 5 requirement is effective pseudocode. The SBS implementation used in this research is very similar to the A* implementation detailed in Section III. In fact, the Python implementation consists of just one class, and this class contains the A*, SBS, and local beam search (see Section V) implementations. As before, then, the pseudocode given here contains only the essential SBS functionality. The reader can find further details in Appendix C.

Like A*, SBS requires a `frontier` set and an `explored` set. Python lists suffice for both of these sets. As previously

Algorithm 2 A Stochastic Approach for MCP/MRP

Input: w, an empty $d \times d$ grid
Input: p, a set of n empty paths
Output: p', a set of n paths
Output: v, the value of p' in w

1: // set of candidates
2: frontier $= \emptyset$
3: explored $= \emptyset$
4: **for** each starting configuration s of n UAVs in w **do**
5: new_world $= w$
6: new_paths $= p$
7: **for** all $u \in$ UAVs **do**
8: new_world[location(u)] $= 1$
9: new_paths $=$ new_paths$_u$ $\cup$ {location(u)}
10: **end for**
11: $v =$ value(new_world, new_paths)
12: frontier $=$ frontier $\cup$ {v, new_world, new_paths}
13: **end for**
14: **while** |frontier| $\neq 0$ **do**
15: // selection
16: $v, w, p =$ frontier.pop_front()
17: explored $=$ frontier $\cup$ {w}
18: // objective
19: **if** $\forall u \in$ UAVs, edge(location(u)) $= 1$ **and** every square is covered or every UAV's battery is depleted **then**
20: // solution
21: **return** (w, p)
22: **end if**
23: // feasibility
24: **for** all $u \in$ UAVs **do**
25: **if** u's battery has 1 unit remaining **then**
26: neighbors $=$ adjacent_edges(u)
27: **else**
28: neighbors $=$ adjacent(u)
29: **end if**
30: **for** all $q \in$ neighbors **do**
31: $w' = w$
32: $p' = p$
33: w'[location(q)] $= 1$
34: $p'_u = p'_u \cup$ location(q)
35: state $=$ {value(w', p'), w', p'}
36: **if** state $\notin$ frontier **and** state $\notin$ explored **then**
37: // next-state generator
38: frontier $=$ frontier $\cup$ {state}
39: **end if**
40: **end for**
41: **end for**
42: // heuristics
43: frontier $=$ frontier.probabilistically_select_best(5)
44: **end while**
45: **return** "No solution found!"

described, each `world` instance (that is, the 2D $d \times d$ grid that contains the sensors and the UAV paths) is contained within a 2D Python list; the path sets are also stored as 2D lists. All other data structures are one of the primitive Python types: `string`, `integer`, `float`, or `boolean` [17].

Pseudocode for the stochastic approach – with the standard search elements labeled with comments – is shown in Algorithm 2.

D. *Implementation*

In this section, we describe Step 6 of the PD/AD process, which requires a one-to-one mapping from Section IV-C pseudocode to an actual, executable implementation. Like we did with the deterministic approach, we decided to implement the SBS function in Python because it an easy-to-write language, it's moderately-fast, and it has a large, well-documented set of tools. We again used PyCharm Professional 2019.1 on a 2017 MacBook Pro to develop the stochastic approach implementation.

This code was written using good software development principles. Our undergraduate- and graduate-level computer science courses in software engineering – and the previously-cited references [20]–[22] – serve as useful softare engineering resources when we need assistance.

Because the SBS implementation is so similar to the A* implementation, the algorithm's complexity is effectively the same as described in Section III-D. Note, however, that the stochastic approach complexity *is* slightly different. Because this SBS implementation probabilistically deletes all but the best $k = 5$ states, the `while-loop` on line 14 is not bounded by $O(8n^2 \times P(d^2, b))$. Instead, it's bounded by $O(8n^2 \times P(d^2, b) \div 5)$, which is certainly very similar. However, Big O notation does not consider constants, so the overall Big O complexity (which is not identical to the exact complexity) is the same as for the A* implementation. We restate that complexity here: $O(n \times P(4d-4, n) + 8n^2 \times P(d^2, b) \div 5) = O(n^2 \times P(d^2, b))$.

The reader can view the full software implementation – including stochastic beam search – in Appendix C.

V. Local Search

Reference [27] says the following of local search:

> "In computer science, local search is a heuristic method for solving computationally hard optimization problems. Local search can be used on problems that can be formulated as finding a solution maximizing a criterion among a number of candidate solutions. Local search algorithms move from solution to solution in the space of candidate solutions (the search space) by applying local changes, until a solution deemed optimal is found or a time bound is elapsed."

In our third attempt to solve MCP/MRP, we employ local beam search (LBS), a local search algorithm much like the previous section's stochastic beam search [15], [26]. Where LBS differs from SBS, however, is in the candidate selection method. As a reminder, SBS *probabilistically* selects the best

k candidates from the `open` set to evaluate at each time step. LBS, on the other hand, *always* selects the best k candidates. The astute reader will recognize that LBS is thus likely to get stuck in local optima without often reaching the global optimum. We expect the same, but effective experimentation principles suggest we attempt to solve MCP/MRP with more than just two approaches.

As in Sections III and IV, we must employ the PD/AD process to effectively develop an LBS algorithm. Like in those sections, we refine the standard search constructs (see Table II) from the algorithm domain (see Section II) to our exact LBS implementation.

The PD/AD process's steps 2–6 for the local approach to MCP/MRP are discussed in the remainder of this section. Details for these steps can be found in Sections V-A, V-B, V-C, and V-D.

A. Design Specification

The PD/AD process's step 2 requires that we select an algorithm domain specification strategy. The standard search constructs for our eventual LBS implementation are defined as follows:

- Set of candidates
  - LBS stores its set of candidates in an `open` list
  - To reduce the search space, we also store *visited* candidates in a `closed` list
- Next-state generator
  - Given an empty world of size $d \times d$, we create $P(4d-4, n)$ initial next-states, one for each starting configuration of n UAVs
  - For a given world, the set of next-states includes all copies of the world with one additional movement of one UAV
    - *Constraint*: the UAV's path length p must be less than b
    - *Constraint*: the UAV cannot move to a non-edge square if $p = b - 1$
- Feasibility
  - For a given state S, determine whether, for every UAV $u \in S$, u is on an edge square and either *a)* remaining_battery(u) $= 0$ or *b)* every square $s \in S$ is covered
  - Determine whether S meets constraints defined in the problem domain
- Selection
  - LBS always selects the k candidates with the lowest $h(S)$ values, where S is the state, and $h(S)$ is the estimate of the remaining cost to the goal
  - Delete those candidates for which $\exists u \in$ UAVs such that remaining_battery(u) $= 0$ and u is not on an edge square
  - Additionally, delete all but the best k candidates
    - To ensure our LBS MCP/MRP implementation is computationally-tractable, let $k = 5$ [28]
- Solution
  - If *feasibility* constraints are met and V_S is minimized, return S
- Objective
  - Minimize $f(S)$ using *heuristics* (defined in Section V-B) while meeting the *feasibility* constraints

B. Design Refinement

The third and fourth steps in the problem domain/algorithm domain process require that one evolves a general solution design specification and instantiates the problem design. We now discuss the abstract data types and structures required in our LBS implementation. We also refine the standard search elements first defined in the previous section.

We know that step 6 of the PD/AD is functional Python code, so we must specify the LBS data structures used to solve MCP/MRP before actually programming. We stated earlier that LBS requires `open` and `closed` sets, so we believe that Python-style lists will prove effective for this research.

As in Sections III-B and IV-B, we now refine the high-level standard search constructs into a more practical form. In doing so, we can more easily map our problem and algorithm domains onto a functioning software implementation.

- Set of candidates
  - open $= \emptyset$
  - closed $= \emptyset$
- Next-state generator
  - If $S = \emptyset$, $\forall c \in$ start_configurations, $S = S \cup \{c\}$
  - Otherwise, $\forall u \in$ UAVs and $\forall a \in$ adjacent_squares$_u$, $S = S \cup \{u \cup \{u_a\}\}$
    - *Constraint*: $|u| \leq b$
    - *Constraint*: If $p = b - 1$, let adjacent_squares$_u$ = adjacent_edge_squares$_u$
- Feasibility
  - Feasible if, $\forall u \in$ UAVs, edge(location(u)) $= 1$ and one of the following holds:
    - remaining_battery(u) $= 0$
    - $\forall s \in S$, covered(s) $= 1$
  - Additionally, each of these problem domain constraints must hold:
    - $|P| = n$; every UAV must have a path (even if the path is empty)
    - $\forall p \in P, |p| \leq b$; no UAV can fly more than b unit squares
    - $\forall p \in P$, $edge(first(p)) = 1$, $edge(last(p)) = 1$; every UAV must start and end on an edge square
    - $\forall Q,\ Q \neq P \Rightarrow V_Q < V_P$; the value of the path set P must be maximal
- Selection
  - At each time t, select up to $k = 5 \arg\max_S h(S)$
  - Delete S if $\exists u \in$ UAVs s.t. remaining_battery(u) $= 0$ and edge(location(u)) $\neq 1$

- Solution
  - Return

$$\begin{cases} S & \text{if feasibility}(S) = 1 \text{ and } V_s \text{ is maximized} \\ \emptyset & \text{otherwise} \end{cases}$$

- Objective
  - Maximize $f(S)$ using *heuristics* while meeting the *feasibility* constraints
  - As defined in the problem domain, $f(S) = v$, the overall state value, which concerns the following:
    * Maximized $\sum_{i=1}^{n} squares\_covered_i$
    * Minimized $\sum_{i=1}^{n} risk_i$
- Heuristics
  - For a given state S, $h(S)$ concerns the following:
    * $total\_ratio = squares\_covered \div d^2$
    * $edge\_ratio = edges\_covered \div (4d - 4)$
    * $same\_repeats$, number of times a UAV visits a square it has already visited
    * $other\_repeats$, number of times a UAV visits a square another UAV has already visited
    * $risk = total\_risk \div path\_length$

C. Pseudocode

We must develop effective pseudocode in step 5 of the PD/AD process. The LBS implementation used in this research is extremely similar to the SBS implementation detailed in Section IV. For this reason, the Python implementation consists of just one class, and this class contains the A*, SBS, and LBS implementations. The pseudocode given here contains only the essential LBS functionality – the various library imports, print statements, and single-value computations are not essential to the reader's understanding of LBS's functionality. Further details of these functions are given in Appendix C.

We know that local beam search requires a `frontier` set and an `explored` set; we believe that Python lists will suffice for these sets. Each `world` instance (that is, the 2D $d \times d$ grid that contains the MCP/MRP sensors and the UAV paths) is contained within a 2D Python list; the path sets are also stored as 2D lists. All other data structures are one of the primitive Python types: `string`, `integer`, `float`, or `boolean` [17].

Pseudocode for the local approach is shown in Algorithm 3. The reader can see that the standard search elements are embedded as comments.

Algorithm 3 A Local Approach for MCP/MRP

Input: w, an empty $d \times d$ grid
Input: p, a set of n empty paths
Output: p', a set of n paths
Output: v, the value of p' in w

// set of candidates
frontier $= \emptyset$
explored $= \emptyset$
for each starting configuration s of n UAVs in w **do**
  new_world $= w$
  new_paths $= p$
  for all $u \in$ UAVs **do**
    new_world[location(u)] $= 1$
    new_paths $=$ new_paths$_u$ $\cup$ {location(u)}
  end for
  $v =$ value(new_world, new_paths)
  frontier $=$ frontier $\cup$ {v, new_world, new_paths}
end for
while |frontier| $\neq 0$ **do**
  // selection
  $v, w, p =$ frontier.pop_front()
  explored $=$ frontier $\cup$ $\{w\}$
  // objective
  if $\forall u \in$ UAVs, edge(location(u)) $= 1$ **and** every square is covered or every UAV's battery is depleted **then**
    // solution
    return (w, p)
  end if
  // feasibility
  for all $u \in$ UAVs **do**
    if u's battery has 1 unit remaining **then**
      neighbors $=$ adjacent_edges(u)
    else
      neighbors $=$ adjacent(u)
    end if
    for all $q \in$ neighbors **do**
      $w' = w$
      $p' = p$
      w'[location(q)] $= 1$
      $p'_u = p'_u \cup$ location(q)
      state $=$ {value(w', p'), w', p'}
      if state $\notin$ frontier **and** state $\notin$ explored **then**
        // next-state generator
        frontier $=$ frontier $\cup$ {state}
      end if
    end for
  end for
  // heuristics
  frontier $=$ frontier.select_best(5)
end while
return "No solution found!"

D. Implementation

Step 6 of the PD/AD process, which requires a one-to-one mapping from Section V-C pseudocode to an actual, executable implementation, is described in this section. Like we did with the other search techniques, we decided to implement the LBS function in Python because it an easy-to-write language, it's moderately-fast, and it has a large, well-documented set of tools. For the third time, we used PyCharm Professional 2019.1 on a 2017 MacBook Pro to develop the search algorithm's implementation.

As before, this code was written using good software development principles. Our undergraduate- and graduate-level computer science courses in software engineering – and the previously-cited references [20]–[22] – serve as useful softare engineering resources when we have questions about effective good enginerring principles.

The LBS implementation is extremely similar to the SBS implementation, so the algorithm's complexity is exactly the same as described in Section IV-D. For clarity, we restate that complexity here: $O(n \times P(4d-4, n) + 8n^2 \times P(d^2, b) \div 5) = O(n^2 \times P(d^2, b))$.

The reader can view the full Python MCP/MRP implementation – including local beam search – in Appendix C.

VI. Experimental Design & Evaluation

To effectively evaluate the various algorithms, we conduct an experiment in which we test each approach on multiple instances of MCP/MRP. This section describes the code and the programming environment, the experimental process, and the results entailed. Additionally, this section discusses how the experiment compares to other experiments found in the literature. We utilize [8]'s "guidelines for testing and reporting" throughout this section to adequately convey all relevant information to the reader.

A. The Computing Environment

The implementation code is fully detailed in Sections III-D, IV-D, and V-D and in Appendix C, so we only briefly describe it here. We implemented two Python3 files: `main.py` and `data.py`. The former contains all algorithmic code; the latter contains only the problem instances used to evaluate the algorithms. Development occurred over a few weeks in April and May of 2019; we slightly revised the code in June 2019.

Pseudocode for the algorithms employed is readily available online [13], [26]; thus, we are confident the algorithms are correctly implemented. Previous experience in software development allows for easy, effective programming, and sufficient documentation means that we were able to revisit the code and ensure its correctness. The debugger in PyCharm, the IDE used to develop this code, allowed us to examine state variables during execution and validate the processes.

All code was written on a 2017 MacBook Pro running macOS Mojave 10.14.4. We used Python version 3.7.3 and PyCharm Professional version 2019.1 to write all of the implementation code. These resources allowed for portability and easy, on-the-go development of the code and of this report.

All tests were conducted within the PyCharm runtime environment. A simple `main` method iterates over the problem instances in `data.py`, and thus one execution of the program can evaluate all 10 problem instances for one algorithm. The user can specific a different algorithm (from the choices *deterministic*, *stochastic*, and *local*) within the `main` function.

All times were measured using Python3's `time` library. This library measures clock cycles of the computer and thus ensures timing accuracy across various systems.

Aside from selecting which algorithm one wishes to evaluate, the code is entirely hands-off. No tuning parameters must be set, and no input is required. Of course, the user can define new problem instances by appending to/modifying the data file.

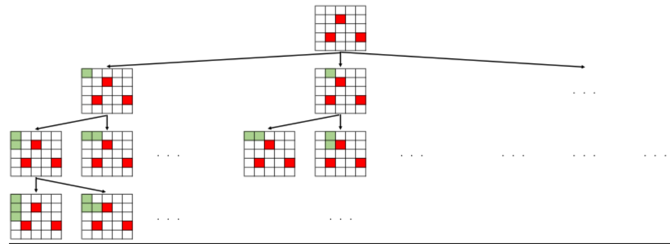

Fig. 2. A potential search tree for a MCP/MRP instance in which $n \geq 1$, $b \geq 3$, $|L| = 3$, $r \approx 0.5$, and $d = 5$. Clearly, the search tree quickly outgrows the available space in this report.

B. The Experimental Results

This research intends to demonstrate whether or not the USAF can feasibly solve MCP/MRP with a search algorithm. If so, the Air Force can guarantee survivability of its UAVs while still conducting sufficient reconnaissance in an area with known sensor locations. If not, the Air Force must reevaluate the problem and perhaps consider other avenues for surveillance.

Of course, the search space for MCP/MRP is exceptionally large. We can see a tree representation of the search space in Fig. 2. A deterministic search, like A*, which must often visit a huge number of the nodes in the tree to reach the globally-optimal solution, simply requires far too much computational time and resources to solve MCP/MRP for all but the smallest of problem instances. Stochastic and local search approaches do not require nearly the same amount of resources, but the sheer size of the search space ensures that neither approach is likely to reach the optimum. However, both the stochastic approach and the local one, if properly tuned, can reach a solution that meets some *good enough* threshold, and both approaches can usually do so in a reasonable amount of time.

The results show this to be the case. We tested each algorithm on 10 different (small) MCP/MRP instances. The exact parameters used for each problem instance can be seen in Table III. Tables III, IV, and V show the execution time required for each of the 10 problems for the deterministic, stochastic, and local approaches, respectively. The tables also display the value of the returned solution for every algorithm-problem instance combination. Clearly, the deterministic approach achieves the highest-valued solutions, but this requires significant execution time; problem 9 even failed to terminate (in 30 minutes or less) for the deterministic approach! However, although the other two algorithmic techniques returned 10 solutions each, most of those solutions are worse than the deterministic approach's solutions. As we can see in figure 3, the stochastic algorithm performed much worse than the other two techniques. Sometimes, the local algorithm performed much like the deterministic approach, but it never performs much better, and it often performs much worse.

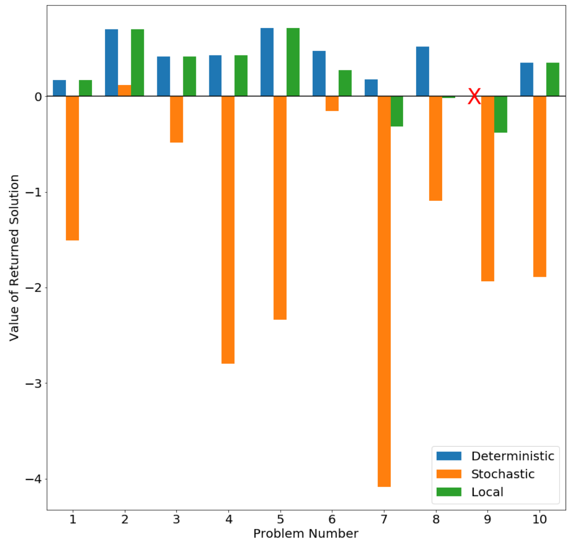

Fig. 3. The results of the experiment. The deterministic algorithm's results are shown in blue; the stochastic algorithm's results are shown in orange; the local algorithm's results are shown in green. The deterministic algorithm did not solve Problem 9.

This illustrates a tradeoff between optimality and resources. This tradeoff is not unexpected, and thus we are not surprised that the fast algorithms perform worse than the slow one.

C. Results Analysis

Results indicate that the algorithms implemented in this research fall into one of two classes:

1) Those that identify strong results but, due to the size of the search space, are unable to terminate for large problem instances; and
2) Those that terminate in a reasonable time but with (usually) poor solutions.

It seems that neither A* nor LBS nor SBS are suitable techniques for solving MCP/MRP, at least where the USAF is concerned. For this reason, the military should consider other multi-domain optimization techniques, including some of those discussed in Appendix B. It's likely that many more approaches – including those that vastly outperform the techniques employed here – are hidden in the literature. Additionally, the USAF best knows its problem domain, so it's likely that those responsible for solving MCP/MRP can better tune the heuristic functions.

VII. Conclusions

In this research, we devised three different approaches to solving MCP/MRP, a multi-domain optimization problem. In doing so, we achieved multiple educational objectives: an understanding of NP-C problem domains; an ability to describe and use various deterministic, nature-inspired, global, and local search algorithms with and without heuristics; and an ability to refine general search algorithms to solve specific problems (among many other objectives met). For this reason, this research is a resounding success.

It's somewhat of a success in other ways, too. Results, as described in Section VI, indicate that an algorithmic approach to MCP/MRP can give feasible solutions for small-to-medium problem instances. Although the algorithms developed in Sections III, IV, and V are not capable as-is of solving practical MCP/MRP instances, it's clear that, with powerful hardware and robust, well-developed heuristics, the USAF may find strong results using one of these methods. Additionally, the experiment conducted is simple enough to allow for easy communication to a layman but complex enough to illustrate the partial success of A*, SBS, and LBS for this problem domain. Some experiments in the literature are certainly more involved than ours, but many are not – for this reason, we assert that ours is successful.

Because we closely followed and documented the PD/AD design process, future researchers can easily reproduce the research presented in this report and utilize the algorithms described to solve their own, similar problems. Additionally, the widespread use of Python ensures that our code will remain relevant for many years to come. Clearly, both the algorithms employed – A*, SBS, and LBS – and the code developed are of a high quality that may prove useful to future students and researchers.

References

[1] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979.
[2] A. Cobham, "The intrinsic computational difficulty of functions," pp. 24–30, 1965.
[3] Wikipedia contributors, "Maximum coverage problem — Wikipedia, the free encyclopedia," 2019. [Online; accessed 04-June-2019].
[4] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, (New York, NY, USA), pp. 151–158, ACM, 1971.
[5] G. B. Lamont, "Lecture 2 'designing solutions for p-time problems'." 03 2019.
[6] Wikipedia contributors, "Problem domain — Wikipedia, the free encyclopedia," 2019. [Online; accessed 06-June-2019].
[7] J. Young and E. Walkingshaw, "A domain analysis of data structure and algorithm explanations in the wild," pp. 870–875, 02 2018.
[8] R. S. Barr, B. L. Golden, J. P. Kelly, M. G. C. Resende, and W. R. Stewart, "Designing and reporting on computational experiments with heuristic methods," *J. Heuristics*, vol. 1, pp. 9–32, 1995.
[9] J. Liu, W. Wang, X. Li, T. Wang, S. Bai, and Y. Wang, "Solving a multi-objective mission planning problem for uav swarms with an improved nsga-iii algorithm," *International Journal of Computational Intelligence Systems*, vol. 11, p. 1067, 05 2018.
[10] S. Çaşka and A. Gayretli, "An algorithm for collaborative surveillance systems with unmanned aerial and ground vehicles," *International Journal of Engineering Trends and Technology*, vol. 33, 04 2016.
[11] G. B. Lamont, J. N. Slear, and K. Melendez, "Uav swarm mission planning and routing using multi-objective evolutionary algorithms," in *2007 IEEE Symposium on Computational Intelligence in Multi-Criteria Decision-Making*, pp. 10–20, April 2007.
[12] Wikipedia contributors, "Deterministic algorithm — Wikipedia, the free encyclopedia," 2019. [Online; accessed 05-June-2019].
[13] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, pp. 100–107, July 1968.
[14] G. B. Lamont, "Admissible h vs. monotonic h in A*." 05 2019.
[15] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2009.

TABLE III
IMPLEMENTATION RESULTS FOR THE DETERMINISTIC SEARCH ALGORITHM

#	n	b	L	r	d	Time (s)	Value
1	2	3	(0.3, 1.8), (3.1, 2.7)	1.2	4	0.064447	0.168319
2	1	3	(0.3, 1.8)	0.3	2	0.001389	0.700000
3	1	4	(1, 1)	3	2	0.001464	0.413514
4	2	7	(0.5, 0.9), (0.6, 1.1)	1.1	3	0.026366	0.426602
5	3	5	(0.5, 1), (1.5, 1), (2.5, 1)	0.5	3	0.063300	0.716667
6	4	2	(0.3, 1.8), (1.1, 2.9), (2.7, 2.7), (3.1, 2.7)	0.3	3	0.297092	0.474632
7	2	7	(-1, -1), (3.5, 2)	3	4	0.147358	0.177236
8	4	3	(0.2, 0.1), (0.3, 0.7), (0.8, 3.4), (3.3, -0.2)	0.6	4	7.490725	0.521721
9	3	4	(1, 1), (3, 1), (4, 1), (4.8, 5.1)	2.1	5	$>$ 30 minutes	Did not finish
10	3	6	(1.60, 0.75), (1.40, 3.65), (3.84, 2.70)	0.75	5	2.728232	0.350254

TABLE IV
IMPLEMENTATION RESULTS FOR THE STOCHASTIC SEARCH ALGORITHM

#	Time (s)	Value
1	0.015266	-1.509787
2	0.000855	0.120098
3	0.001523	-0.486486
4	0.024772	-2.799535
5	0.041905	-2.337500
6	0.012181	-0.152573
7	0.051217	-4.085834
8	0.050550	-1.095355
9	0.110520	-1.936070
10	0.075914	-1.892918

TABLE V
IMPLEMENTATION RESULTS FOR THE LOCAL SEARCH ALGORITHM

#	Time (s)	Value
1	0.009594	0.168318
2	0.000691	0.700000
3	0.000796	0.413513
4	0.012163	0.426602
5	0.011291	0.716666
6	0.017952	0.271139
7	0.026866	-0.313384
8	0.060396	-0.018886
9	0.112361	-0.382878
10	0.149047	0.350254

[16] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1984.

[17] Python Software Foundation, "Built-in types," 2019. [Online; accessed 05-June-2019].

[18] JetBrains s.r.o., "Pycharm: The python ide for professional developers," 2019. [Online; accessed 05-June-2019].

[19] Apple, Inc., "Macbook pro," 2019. [Online; accessed 05-June-2019].

[20] I. Sommerville, *Software Engineering*. USA: Addison-Wesley Publishing Company, 9th ed., 2010.

[21] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 ed., 1994.

[22] H. Gomaa, *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*. Cambridge University Press, 2011.

[23] Wikipedia contributors, "Permutation — Wikipedia, the free encyclopedia," 2019. [Online; accessed 05-June-2019].

[24] Wikipedia contributors, "Stochastic optimization — Wikipedia, the free encyclopedia," 2019. [Online; accessed 06-June-2019].

[25] J. Brownlee, *Clever Algorithms: Nature-Inspired Programming Recipes*. Lulu.com, 1st ed., 2011.

[26] Wikipedia contributors, "Beam search — Wikipedia, the free encyclopedia," 2019. [Online; accessed 06-June-2019].

[27] Wikipedia contributors, "Local search (optimization) — Wikipedia, the free encyclopedia," 2019. [Online; accessed 06-June-2019].

[28] Wikipedia contributors, "Computational complexity theory — Wikipedia, the free encyclopedia," 2019. [Online; accessed 06-June-2019].

[29] Wikipedia contributors, "List of np-complete problems year =."

[30] J. Kleinberg and E. Tardos, *Algorithm Design*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2005.

[31] D. S. Hochbaum, "Approximation algorithms for np-hard problems," ch. Approximating Covering and Packing Problems: Set Cover, Vertex Cover, Independent Set, and Related Problems, pp. 94–143, Boston, MA, USA: PWS Publishing Co., 1997.

[32] D. P. Williamson and D. B. Shmoys, *The Design of Approximation Algorithms*. New York, NY, USA: Cambridge University Press, 1st ed., 2011.

[33] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *Trans. Evol. Comp*, vol. 1, pp. 67–82, Apr. 1997.

[34] E. F Moore, "The shortest path through a maze," pp. 285–292, 01 1959.

[35] G. B. Lamont, "3.4 global search - breath first search." 04 2019.

[36] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artif. Intell.*, vol. 27, pp. 97–109, Sept. 1985.

[37] Wikipedia contributors, "Iterative deepening A* — Wikipedia, the free encyclopedia," 2018. [Online; accessed 07-June-2019].

[38] C. A. C. Coello, G. B. Lamont, and D. A. V. Veldhuizen, *Evolutionary Algorithms for Solving Multi-Objective Problems (Genetic and Evolutionary Computation)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[39] E.-G. Talbi, *Metaheuristics: From Design to Implementation*. John Wiley & Sons, 2009.

[40] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *SCIENCE*, vol. 220, no. 4598, pp. 671–680, 1983.

[41] F. Glover, "Future paths for integer programming and links to artificial intelligence," *Comput. Oper. Res.*, vol. 13, pp. 533–549, May 1986.

[42] F. Moyson, B. Manderick, and V. U. B. A. I. Laboratory, *The Collective*

Behavior of Ants: An Example of Self-organization in Massive Parallelism. AI memo, Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1988.

[43] G. B. Lamont, "Swarm intelligence: Part I: The origins of ant colony optimization techniques." 2016.

Appendix A
Proof of Complexity

One goal for this research is that we develop approximation algorithms for difficult, multi-domain optimization problems. To ensure that MCP/MRP is *difficult* enough, we show here that both MCP and MRP are NP-C problems.

The maximum coverage problem portion of MCP/MRP is similar to each of the following NP-C problems: bin-packing, traveling salesman decision, and vertex cover [29]. One can reduce MCP – which is exactly the problem of routing n UAVs (of capacity b) through a $d \times d$ grid to maximize coverage – to any of these problems (and to any other NP-C problem [30]). Additionally, these NP-C are reducible to MCP. Thus, MCP is itself NP-C [3], [31].

The minimum risk problem portion of MCP/MRP concerns the problem of routing an agent through a $d \times d$ grid without visiting penalty squares. For this reason, we assert that MRP is similar to the subset sum, graph coloring, and graph partition problems, all of which are NP-C [29], and each of which penalizes the algorithm for incorrectly visiting/coloring adjacent nodes. Because MRP is polynomial-time reducible to any of these NP-C problems, and because all are reducible to MRP, we claim that MRP is also NP-C [30].

Because both MCP and MRP are NP-C problems, we absolutely cannot guarantee an optimal solution to all problem instances with a deterministic, polynomial-time algorithm. This is why we developed various approximation algorithms in an attempt to solve MCP/MRP [32]. It is certainly possible that the algorithms we implemented – A*, stochastic beam search, and LBS – are not suitable to solving MCP/MRP, so we discuss other algorithms in the following Appendices.

Appendix B
Other Search Techniques

The "no free lunch" theorem "state[s] that any two optimization algorithms are equivalent when their performance is averaged across all possible problems" [33]. However, we know that MCP/MRP is an exceedingly difficult problem, so are not concerned with *all possible problems*; rather, we are concerned with small-to-medium problem instances. For this reason, it's possible that some variations on the search techniques already described may perform better than the techniques themselves.

In this Appendix, then, we describe potentially-useful variations on the algorithms used to solve MCP/MRP in Sections III, IV, and V. We present these variations on A*, on stochastic beam search, and on local beam search in Appendices B-A, B-B, and B-C, respectively. Additionally, we discuss in Appendix B-D why an insect-inspired search algorithm may or may not be useful for this problem.

A. Deterministic Search

We showed that A*, itself a variation of breadth-first search (via informedness) [16], [34], [35], is somewhat successful at solving MCP/MRP. However, the problem's search space is simply too large to fully utilize standard A*; for this reason, we believe iterative deepening A* (IDA*) can sufficiently restrict the search space to allow for better results [36], [37]. Because the heuristic seeks to maximize the UAV coverage of the surveillance area, and because each new level of the tree indicates that one UAV has moved one square, A* is likely to explore to extreme depths before exploring the width of the tree. By iteratively bounding the exploration depth with IDA*, we expect to identify *sufficient* solutions to MCP/MRP. Although this won't guarantee the optimality of a given solution, it does guarantee that the search finishes in a reasonable amount of time. Of course, large values for n significantly increase the search tree's branching factor, so IDA* won't *always* perform better than A* (see [33] for a discussion on the "no free lunch" theorem).

B. Stochastic Search

The stochastic search we implemented, stochastic beam search, worked for our purposes (that is, problem demonstration and solution feasibility). However, it's not good enough for the *United States Air Force's* purposes. It's possible that a different stochastic search algorithm outperforms SBS in solving MCP/MRP. For this reason, we turn to genetic algorithms (GAs). Genetic algorithms utilize stochasticity to solve difficult problems by emulating evolutionary biology [38], [39]. Because GAs build a population of solutions (individuals) by combining high-value solution candidates, and because the MCP/MRP search space contains numerous possible paths, a GA approach is likely to work well in solving this problem. Of course, we must develop well-tuned selection, combination, and mutation operators, but this is possible through metaheuristic tuning [39]. In doing so, we can utilize a GA algorithm to solve MCP/MRP, and we expect results to improve upon those given by SBS.

C. Local Search

Of the three algorithms implemented to solve MCP/MRP, local beam search is the weakest. LBS is not able to escape locally-optimal solutions, so it fails to find the optimum (or close to the optimum) as often as needed. Perhaps another local search algorithm can give better performance than LBS. Specifically, simulated annealing (SA), which *can* escape local optima, is likely to outperform LBS. SA is a well-known local search technique, but it avoids many of the pitfalls of other local techniques while still reaching the globally-optimal solution in many cases [39], [40]. With a slow enough cooling schedule, SA will certainly improve on LBS's MCP/MRP results – and it might also improve on the deterministic and stochastic approach results, too.

Of course, other local search algorithms might prove better for our desired problem instances (again, see [33]'s discussion of the "no free lunch" theorem). Tabu search is one other local search technique that might work well because it too can escape local optima [41]. Further research is required to determine the effectiveness of Tabu search in solving MCP/MRP.

D. Insect-Inspired Search

In this research, we implemented one deterministic search algorithm, one stochastic search algorithm, and one local search algorithm. However, other search techniques are available. Specifically, the field of insect-inspired search algorithms includes numerous techniques that might apply to MCP/MRP. One such technique is ant colony optimization (ACO), a search algorithm that utilizes a collection of ants, each of which returns a solution candidate at every time step [42], [43]. As discussed in Appendix B-B, the huge number of potential paths through a $d \times d$ grid can be represented by a large population of solution candidates. Like in a GA approach, an ACO approach can construct a large population and guide the search (that is, the ants) towards an optimal solution. The pheromones, of course, must be finely-tuned with a hyperparameter evaluation scheme to ensure the ACO algorithm can reach the best possible solution [43].

Appendix C
Implemented Code

We now present a representation of the input file, `data.py`. In general, this file should contain all problem instances one wishes to test, but the reader should have an idea of the structure of each problem instance by examining this code.

```
problems = [
    {
        "size": 4,
        "drones": 2,
        "battery": 3,
        "radius": 1.2,
        "locations": [(0.3,1.8),(3.1,2.7)]
    },

    ...

    {
        "size": 5,
        "drones": 3,
        "battery": 4,
        "radius": 2.1,
        "locations": [(1,1),(3,1),(4,1),(4.8,5.1)]
    },
]
```

In the remainder of this appendix, the reader may view information about the entire Python codebase. Much of the `MaxCoverageMinRisk` class functions serve only to output data or to compute various metrics for the deterministic, stochastic, and local search techniques. Still, the reader may find the different sections in the locations given in Table VI.

TABLE VI
LINE NUMBER RANGES FOR VARIOUS CODE SECTIONS

Section	Line Numbers
Initialization	8–13, 58–74
Visual output	15–56
Helper tools	76–165
Risk evaluation	167–195
State evaluation	197–256
Deterministic search	300–411
Stochastic search	258–284, 413–521
Local search	286–298, 523–631
Testing	633–666

Although the standard header format is not present in the code itself, Table VII presents the contents of such a header.

TABLE VII
STANDARD HEADER DETAILS FOR `MaxCoverageMinRisk`

Header Item	Details
Title	Maximum Coverage – Minimum Risk
Date	7 June 2019
Version	1.1
Project	CSCE 686 Multi-Domain Optimization
Author	2d Lt David Crow
Problem domain	NP-C$^+$
Design process	Problem domain/algorithm domain design process
Abstract data types	Lists (including 2D lists), sets, classes, primitives
Algorithm	A*, stochastic beam search, stochastic beam search
Algorithm complexity	$O(n^2 \times P(d^2, b))$ for each algorithm
Operating system	macOS Mojave, Version 10.14.4
Language	Python 3.7.3
Globals	None
Parameters	input: n, b, L, r, d; output: p', $v = value(p')$
Local variables	`frontier`, `explored`, `world`, `paths`, `value`
Modules	`main`, `data`
Imports	`numpy`, `copy.deepcopy`, `itertools.combinations`, `data.problems`
Files	`main.py`, `data.py`
History/revisions	1.0: individual algorithms developed over two weeks in Sp2019; 1.1: combined algorithms into one project

The full `main.py`, which contains all algorithms and overhead, is shown below.

```
1  import numpy as np
2  from copy import deepcopy
3  from data import problems
4  from itertools import combinations
5
6
7  class MaxCoverageMinRisk:
8      def __init__(self, instance):
9          self.size = instance["size"]
10         self.drones = instance["drones"]
11         self.battery = instance["battery"]
12         self.radius = float(instance["radius"])
13         self.locations = instance["locations"]
14
15     # pretty-print a 2D world
16     @staticmethod
17     def show_world(world):
18         for row in world:
19             for i in row:
20                 print("x" if i == 1 else "-", end="\t")
21             print()
22         print()
23
24     # print all path worlds side-by-side
25     def show_paths(self, paths):
26         path_worlds = []
27
28         # build the path worlds
```

```
        for path in paths:
            pw = self.init_world()

            for i in range(len(path)):
                self.cover(pw, path[i][0], path[i][1], i + 1)

            path_worlds.append(pw)

        # print the i-th row of every path world (with separators)
        for i in range(self.size):
            for pw in path_worlds:
                for j in pw[i]:
                    print(j, end="\t")
                if pw != path_worlds[-1]:
                    print("|", end="\t")
            print()

        print()

    # pretty print a path world
    def show_path(self, path):
        path_world = self.init_world()

        # build the path world
        for i in range(len(path)):
            self.cover(path_world, path[i][0], path[i][1], i + 1)

        self.show_world(path_world)

    # build a size * size world of all zeroes
    def init_world(self):
        world = []
        for i in range(self.size):
            world.append([])
            for j in range(self.size):
                world[i].append(0)

        return world

    # give every drone an empty path
    def init_paths(self):
        paths = []
        for i in range(self.drones):
            paths.append([])

        return paths

    # return the world value of (x, y)
    def at(self, world, x, y):
        return world[self.size - y - 1][x]

    # give (x, y) a value (usually 1)
    def cover(self, world, x, y, value=1):
        world[self.size - y - 1][x] = value

    # returns the total number of covered squares
    def total_covered(self, world):
        total = 0

        for i in range(self.size):
            for j in range(self.size):
                total += self.at(world, i, j)

        return total

    # returns the total number of covered edges
    def edges_covered(self, world):
        total = 0

        # count the edges
        for i in range(self.size):
            total += self.at(world, i, 0)
            total += self.at(world, 0, i)
            total += self.at(world, i, self.size - 1)
            total += self.at(world, self.size - 1, i)

        # remove duplicates (corners)
```

```
        for i in (0, self.size - 1):
            for j in (0, self.size - 1):
                total -= self.at(world, i, j)

        return total

    # returns a set of all edges
    def edges(self):
        edge_set = []

        # add all non-corner edges to edge_set
        for i in range(self.size - 2):
            edge_set.append((i + 1, 0))
            edge_set.append((0, i + 1))
            edge_set.append((i + 1, self.size - 1))
            edge_set.append((self.size - 1, i + 1))

        # add all corners to edge_set
        for i in (0, self.size - 1):
            for j in (0, self.size - 1):
                edge_set.append((i, j))

        return edge_set

    # returns the (up to) eight squares adjacent to (x, y)
    def adjacent(self, x, y):
        possible_x = [x]
        possible_y = [y]

        neighbors = []

        # checks whether we're on an edge
        if x < self.size - 1:
            possible_x.append(x + 1)
        if x > 0:
            possible_x.append(x - 1)
        if y < self.size - 1:
            possible_y.append(y + 1)
        if y > 0:
            possible_y.append(y - 1)

        for i in possible_x:
            for j in possible_y:
                neighbors.append((i, j))

        # the current square is not a neighbor of the current square
        neighbors.remove((x, y))
        return neighbors

    # returns the edge squares adjacent to (x, y)
    def adjacent_edges(self, x, y):
        neighbors = self.adjacent(x, y)
        edge_set = deepcopy(neighbors)

        # if it's not an edge cell, remove it
        for n in neighbors:
            if (n[0], n[1]) not in self.edges():
                edge_set.remove(n)

        return edge_set

    # return the risk an (x, y) square contains
    def calculate_risk(self, square):
        risk = 0

        # sum all risk values for this square
        for l in self.locations:
            # find the coordinates of intersection
            left = max(square[0], l[0] - self.radius)
            right = min(square[0] + 1, l[0] + self.radius)
            bottom = max(square[1], l[1] - self.radius)
            top = min(square[1] + 1, l[1] + self.radius)

            # if the rectangle exists, return the ratio defined by rectangle / square
            if left < right and bottom < top:
                total_area = 1 + 4 * self.radius * self.radius
                intersecting_area = total_area - (right - left) * (top - bottom)
```

```
                risk += intersecting_area / total_area

        return risk

    # return the risk a UAV in (x, y) gains
    def risk_value(self, path):
        risk = 0

        # compute risk value for each square
        for p in path:
            risk += self.calculate_risk(p)

        return risk / len(path)

    ######################################
    ### STANDARD SEARCH ELEMENT: HEURISTIC
    ######################################

    # estimate the value of a given world
    def value(self, world, paths, show_details=False):
        # we want to cover as many squares as possible
        total_ratio = self.total_covered(world) / float(self.size * self.size)

        # we should leave the edges open for later exit
        edge_ratio = 1 - (self.edges_covered(world) / float(4 * self.size - 4))

        # count the number of repeat visits to any given square
        same_repeats = 0
        other_repeats = 0
        for path in paths:
            # find every visited square for all other paths
            all_locations = []

            for p in paths:
                if p is not path:
                    all_locations += p

            # if the current path visits one of those squares, +1 repeat
            for p in path:
                if p in all_locations:
                    other_repeats += 1

            for i in range(len(path)):
                for j in range(i + 1, len(path)):
                    if path[i] == path[j]:
                        same_repeats += 1

        # these are double counted in the loop above
        other_repeats = int(other_repeats / 2.0)

        # compute every UAV's risk value
        risk = 0

        for path in paths:
            risk += self.risk_value(path)

        risk /= len(paths)

        # apply weights to the various metrics
        value = 0.90 * total_ratio \
            + 0.10 * edge_ratio    \
            - 0.25 * other_repeats \
            - 0.25 * same_repeats  \
            - 0.50 * risk

        if show_details:
            print("Total ratio    :", total_ratio)
            print("Edge ratio     :", edge_ratio)
            print("Other repeats :", other_repeats)
            print("Same repeats   :", same_repeats)
            print("Maximum risk   :", risk)
            print("State value    :", value)

        return value

    ######################################
    ### STANDARD SEARCH ELEMENT: SELECTION
```

```
    #####################################

    # remove all but the best k states from the frontier
    @staticmethod
    def stochastic_filter(frontier, explored, k):
        indices = np.arange(len(frontier))

        # build a probability mapping
        weights = []
        for f in frontier:
            weights.append(f[0])

        weights = np.square(np.array(weights))

        # normalize them
        weights = weights / weights.sum()

        # probabilistically select the best k weights
        indices = list((np.random.choice(indices, size=k, p=weights)))

        # update the frontier and the explored set
        for i in range(len(frontier) - 1, -1, -1):
            if i not in indices:
                explored.append(frontier[i])
                del frontier[i]

    #####################################
    ### STANDARD SEARCH ELEMENT: SELECTION
    #####################################

    # remove all but the best k states from the frontier
    @staticmethod
    def local_filter(frontier, explored, k):
        frontier.sort(reverse=True)

        # update the frontier and the explored set
        for i in range(len(frontier) - 1, k - 1, -1):
            explored.append(frontier[i])
            del frontier[i]

    # deterministic search
    def a_star(self, w, p):

        #############################################
        ### STANDARD SEARCH ELEMENT: SET OF CANDIDATES
        #############################################

        frontier = []
        explored = []

        ###############################################
        ### STANDARD SEARCH ELEMENT: NEXT-STATE GENERATOR
        ###############################################

        # add every starting configuration to the frontier
        configurations = combinations(self.edges(), self.drones)
        for config in configurations:
            new_world = deepcopy(w)
            new_paths = deepcopy(p)

            # start the drones in their designated spots
            for i in range(self.drones):
                self.cover(new_world, config[i][0], config[i][1])
                new_paths[i].append(config[i])

            frontier.append((self.value(new_world, new_paths), new_world, new_paths))

        # while we still have states to search
        while len(frontier) > 0:

            #####################################
            ### STANDARD SEARCH ELEMENT: OBJECTIVE
            #####################################

            # pull the next state from the frontier
            frontier.sort(reverse=True)
            _, world, paths = frontier.pop(0)
```

```
            explored.append(world)

            ########################################
            ### STANDARD SEARCH ELEMENT: FEASIBILITY
            ########################################

            # if the world is covered and every drone is on an edge square
            goal_found = True
            if self.total_covered(world) == self.size * self.size:
                for path in paths:
                    current = path[-1]
                    if current[0] != 0                    \
                        and current[0] != self.size - 1   \
                        and current[1] != 0               \
                        and current[1] != self.size - 1:

                        goal_found = False

            ########################################
            ### STANDARD SEARCH ELEMENT: FEASIBILITY
            ########################################

            # otherwise, if every drone has depleted its battery
            else:
                for path in paths:
                    # if at least one drone has remaining battery capacity
                    if len(path) < self.battery:
                        goal_found = False
                        break

            #####################################
            ### STANDARD SEARCH ELEMENT: SOLUTION
            #####################################

            if goal_found:
                return world, paths

            ################################################
            ### STANDARD SEARCH ELEMENT: NEXT-STATE GENERATOR
            ################################################

            # for every drone...
            for i in range(self.drones):
                # drone's current location
                current = paths[i][-1]

                # find the adjacent nodes (consider strictly edge nodes if the battery is nearly depleted)
                neighbors = None
                if len(paths[i]) == self.battery - 1:
                    neighbors = self.adjacent_edges(current[0], current[1])
                elif len(paths[i]) < self.battery - 1:
                    neighbors = self.adjacent(current[0], current[1])

                # if the drone can move
                if neighbors is not None:
                    # for every adjacent square...
                    for neighbor in neighbors:
                        # move the drone to the square
                        new_world = deepcopy(world)
                        self.cover(new_world, neighbor[0], neighbor[1])

                        # update the paths
                        new_paths = deepcopy(paths)
                        new_paths[i].append(neighbor)

                        ######################################
                        ### STANDARD SEARCH ELEMENT: OBJECTIVE
                        ######################################

                        # collect the new state's information
                        new_state = (self.value(new_world, new_paths) - 1, new_world, new_paths)

                        # if we've never seen the state, add it to the frontier
                        if new_state not in frontier and new_state not in explored:
                            frontier.append(new_state)

    # stochastic search
```

```
    def stochastic_beam_search(self, w, p):

        ###############################################
        ### STANDARD SEARCH ELEMENT: SET OF CANDIDATES
        ###############################################

        frontier = []
        explored = []

        ##################################################
        ### STANDARD SEARCH ELEMENT: NEXT-STATE GENERATOR
        ##################################################

        # add every starting configuration to the frontier
        configurations = combinations(self.edges(), self.drones)
        for config in configurations:
            new_world = deepcopy(w)
            new_paths = deepcopy(p)

            # start the drones in their designated spots
            for i in range(self.drones):
                self.cover(new_world, config[i][0], config[i][1])
                new_paths[i].append(config[i])

            frontier.append((self.value(new_world, new_paths), new_world, new_paths))

        # while we still have states to search
        while len(frontier) > 0:
            # pull the next state from the frontier
            _, world, paths = frontier.pop(0)
            explored.append(world)

            ########################################
            ### STANDARD SEARCH ELEMENT: FEASIBILITY
            ########################################

            # if the world is covered and every drone is on an edge square
            goal_found = True
            if self.total_covered(world) == self.size * self.size:
                for path in paths:
                    current = path[-1]
                    if current[0] != 0                  \
                        and current[0] != self.size - 1 \
                        and current[1] != 0             \
                        and current[1] != self.size - 1:

                        goal_found = False

            ########################################
            ### STANDARD SEARCH ELEMENT: FEASIBILITY
            ########################################

            # otherwise, if every drone has depleted its battery
            else:
                for path in paths:
                    # if at least one drone has remaining battery capacity
                    if len(path) < self.battery:
                        goal_found = False
                        break

            #####################################
            ### STANDARD SEARCH ELEMENT: SOLUTION
            #####################################

            if goal_found:
                return world, paths

            ##################################################
            ### STANDARD SEARCH ELEMENT: NEXT-STATE GENERATOR
            ##################################################

            # for every drone...
            for i in range(self.drones):
                # drone's current location
                current = paths[i][-1]

                # find the adjacent nodes (consider strictly edge nodes if the battery is nearly depleted)
```

```
                neighbors = None
                if len(paths[i]) == self.battery - 1:
                    neighbors = self.adjacent_edges(current[0], current[1])
                elif len(paths[i]) < self.battery - 1:
                    neighbors = self.adjacent(current[0], current[1])

                # if the drone can move
                if neighbors is not None:
                    # for every adjacent square...
                    for neighbor in neighbors:
                        # move the drone to the square
                        new_world = deepcopy(world)
                        self.cover(new_world, neighbor[0], neighbor[1])

                        # update the paths
                        new_paths = deepcopy(paths)
                        new_paths[i].append(neighbor)

                        ######################################
                        ### STANDARD SEARCH ELEMENT: OBJECTIVE
                        ######################################

                        # collect the new state's information
                        new_state = (self.value(new_world, new_paths) - 1, new_world, new_paths)

                        # if we've never seen the state, add it to the frontier
                        if new_state not in frontier and new_state not in explored:
                            frontier.append(new_state)

            # probabilistically select the best k states
            self.stochastic_filter(frontier, explored, 5)

    # local search
    def local_beam_search(self, w, p):

        ##############################################
        ### STANDARD SEARCH ELEMENT: SET OF CANDIDATES
        ##############################################

        frontier = []
        explored = []

        ##################################################
        ### STANDARD SEARCH ELEMENT: NEXT-STATE GENERATOR
        ##################################################

        # add every starting configuration to the frontier
        configurations = combinations(self.edges(), self.drones)
        for config in configurations:
            new_world = deepcopy(w)
            new_paths = deepcopy(p)

            # start the drones in their designated spots
            for i in range(self.drones):
                self.cover(new_world, config[i][0], config[i][1])
                new_paths[i].append(config[i])

            frontier.append((self.value(new_world, new_paths), new_world, new_paths))

        # while we still have states to search
        while len(frontier) > 0:
            # pull the next state from the frontier
            _, world, paths = frontier.pop(0)
            explored.append(world)

            ########################################
            ### STANDARD SEARCH ELEMENT: FEASIBILITY
            ########################################

            # if the world is covered and every drone is on an edge square
            goal_found = True
            if self.total_covered(world) == self.size * self.size:
                for path in paths:
                    current = path[-1]
                    if current[0] != 0                   \
                        and current[0] != self.size - 1  \
                        and current[1] != 0              \
```

```
                        and current[1] != self.size - 1:

                        goal_found = False

            ########################################
            ### STANDARD SEARCH ELEMENT: FEASIBILITY
            ########################################

            # otherwise, if every drone has depleted its battery
            else:
                for path in paths:
                    # if at least one drone has remaining battery capacity
                    if len(path) < self.battery:
                        goal_found = False
                        break

            #####################################
            ### STANDARD SEARCH ELEMENT: SOLUTION
            #####################################

            if goal_found:
                return world, paths

            #################################################
            ### STANDARD SEARCH ELEMENT: NEXT-STATE GENERATOR
            #################################################

            # for every drone...
            for i in range(self.drones):
                # drone's current location
                current = paths[i][-1]

                # find the adjacent nodes (consider strictly edge nodes if the battery is nearly depleted)
                neighbors = None
                if len(paths[i]) == self.battery - 1:
                    neighbors = self.adjacent_edges(current[0], current[1])
                elif len(paths[i]) < self.battery - 1:
                    neighbors = self.adjacent(current[0], current[1])

                # if the drone can move
                if neighbors is not None:
                    # for every adjacent square...
                    for neighbor in neighbors:
                        # move the drone to the square
                        new_world = deepcopy(world)
                        self.cover(new_world, neighbor[0], neighbor[1])

                        # update the paths
                        new_paths = deepcopy(paths)
                        new_paths[i].append(neighbor)

                        ######################################
                        ### STANDARD SEARCH ELEMENT: OBJECTIVE
                        ######################################

                        # collect the new state's information
                        new_state = (self.value(new_world, new_paths) - 1, new_world, new_paths)

                        # if we've never seen the state, add it to the frontier
                        if new_state not in frontier and new_state not in explored:
                            frontier.append(new_state)

            # select the best k states
            self.local_filter(frontier, explored, 5)

def main():
    method = "deterministic"
    # method = "stochastic"
    # method = "local"

    for i, problem in enumerate(problems):
        print("PROBLEM INSTANCE {}:\t".format(i + 1))
        for key in problem:
            print("\t{} : {}".format(key, problem[key]))

        print()
        solver = MaxCoverageMinRisk(problem)
```

```
        world = None
        paths = None

        if method == "deterministic":
            world, paths = solver.a_star(solver.init_world(), solver.init_paths())
        elif method == "stochastic":
            world, paths = solver.stochastic_beam_search(solver.init_world(), solver.init_paths())
        elif method == "local":
            world, paths = solver.local_beam_search(solver.init_world(), solver.init_paths())

        print("World\n-----")
        solver.show_world(world)

        print("Paths\n-----")
        solver.show_paths(paths)

        print("Value\n-----")
        solver.value(world, paths, show_details=True)

        if i != len(problems) - 1:
            print("\n")


main()
```