EC ENGR 134, Fall 2021: Kellen Cheng (905155544) and Yu-Wei Vincent Yeh (005123289)
Sensor Network Synthesis - Part 2: Drone Deployment
Due: 5 December, 2021 at 10:00 PM

1 Problem Statement

The study of wireless sensor networks has drawn great interests within disciplines such as area monitoring, environmental detection, and space defense. Through an optimal placement of sensors, environmental obstructions of the signal could be mitigated while maintaining good connectivity and coverage - the connectivity and coverage that arise from this optimal placement are the critical figures of merits to this type of study. In part 1 of this project, we have devised a threshold sensor placement algorithm that not only ensured strong connectivity, but also achieved a supreme coverage, reaching 90% with only 1250-1300 sensors - our sensor allocation was posted on CCLE, and adopted as the basis of part 2 for the entire class. The goal of this part of the project is to design a trajectory algorithm for the drones to drop sensors to the designated locations under two scenarios: 1) One large drone that traverses through the map, dropping sensors within 5x5 squares, and 2) M small drones delivering sensors simultaneously that minimizes delivery time, under fuel and sensor carrying constraints. Built on our previous success from part one, we created a highly generalizable routing algorithm that produces highly efficient routes using the minimum set cover, and a state of the art polar-coordinate-based radial partitioning algorithm that aids in creating minimal-loss paths for a multi-drone traversal under extraneous constraints. Unlike the allocation-specific path finding algorithms others have developed, that may work for one distribution but fail in many other cases, our multifaceted algorithm was developed without prior assumptions on any specific distribution.

2 Algorithm Description

Overview: Since the ultimate goal is to find a short cycle that covers the coordinates of all sensors, we would like to first figure out which sections of the sensor distribution has the highest density of sensors, which is the responsibility of the functions described in 2.1. After finding the "semi-squares" (sets) that covers all sensor coordinates with the functions in 2.1, we would like to connect the segments to one another in an efficient manner, and in the case of multi-drone delivery, we would like to partition the minimum set segments, which are the responsibilities of functions described in 2.2.

2.1 Path Finding & Minimum Set Cover Algorithms:

(a) **square_cycles_size_n**: The goal of this function is to output a dictionary that maps the list of sensor locations covered to every possible three-sided square on the grid, that would later be used in the minimum set cover algorithm. This algorithm takes in three parameters: n, the side length of the arbitrary squares, grid, the map that the sensor is on, and all_loc, the location of all sensors. The side length n acts as a hyper-parameter for fine tuning in order to achieve the shortest cycle length, and it was determined that n=6 produces an optimal result. The reason to use only three sides of squares is because to create a cycle, an entry and exit point is required for the line segments. However, if we want to traverse a full square with 4 sides, determining where to enter and where to exit would create unnecessary complexities. Furthermore, using consecutive sides not only allowed us to string the segments into a path more easily, the computational cost associated with the "stringing" is also extremely low.

(b) **min_set_cover**: The goal of this function is to output a dictionary that gives the minimum set cover based on the parameters: cover_dict, the dictionary that contains some coordinates as keys and sensor locations covered as coordinates, and all_loc, the locations of sensors that need to be covered. To obtain the min set cover, we repeatedly find the elements in cover_dict that have the greatest intersection with the sensors that we want to eliminate, and over each iteration, we remove the sensors already covered, to greedily maximize the overlap of the new sets added.

(c) **edges_path**: Since we previously based our min set cover problem on the arbitrary three-sided "semi-squares", some sides in the group of edges might not actually cover any sensors, so this function intends to take out these edges, and return a new dictionary that has edges as keys, and the list of sensors eliminated as values.

2.2 Path Construction & Partition Algorithms:

(d) **edge_connecting_algorithm**: This algorithm employs a forward-back and back-forward stacking, reverse augmented binary search approach to construct a cycle based on the inputs: the center location in which the drone starts and ends, and the edge set determined by the minimum set cover algorithm that that we want to string together. The algorithm first goes through every edge in the set cover to combine edges that connect with each other into sets, as they should be in our original construct of the "semi-squares." Then, the algorithm produces a cycle that starts with center_loc and ends with center_loc by repeatedly finding the 1st and 2nd closest set of edges to the latest set that we added to the path array, such that we build the path concurrently in the forward and backward direction. The end product would take the form [(150,150),closest1,closest1',.....,closest2',closest2,(150,150)]. Since this algorithm is somewhat greedy, towards the end, the "closest set" became divergent. However, as shown in the report, the divergent behavior towards the end is insignificant, with respect to the distance already traveled. Finally, the algorithm returns a collection of sets of edges (the three sides of the incomplete square, assuming no edges was eliminated), and the true path that contains all sets of edges that the drone traverses.

(e) **partition_algorithm**: In the case of a multi-drone delivery system, a partition of the plane is required. Since the drones eventually have to complete a cycle by coming back to the sink, we propose a polar-coordinate-based radial partitioning algorithm centered around the sink, such that each slice of the plane would have similar number of tasks. The algorithm begins with converting the coordinate of each minimum set edge into polar coordinates, stored in a dictionary with edges being the key, and the angles, which take values from 0 to 2pi, being the values. Then, the edges are sorted based on the angles, then partitioned into equal parts based on the number of sections we would like to partition. The end result is a list of dictionaries that contains the edges in each slice of the plane.

(f) **path_length_calculation**: This function intends to calculate the path length given the path of a cycle. This function takes the following inputs: y, the path generated by the edge_connecting_algorithm, edge_grid, the array we would like to visualize the trajectory on, n, the side length of our arbitrary squares, and all_loc, the location of some sensors in the path. Using distance-calculating utility functions, dist_pt_sets and dist_PtoS, this function repeatedly adds up the distance between two sets of edges, and outputs the length of the cycle.

2.3 Utility Functions:

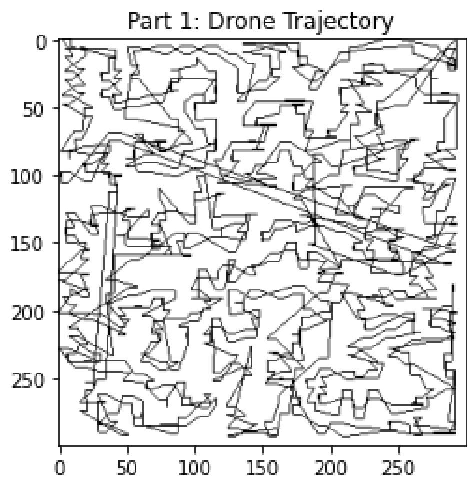
(g) **num_sensors_eliminated**: With each drone covering a 5x5 square at any instant, flying from one location to another would result in a channel-like shape. Thus, this utility function looks for sensors within the boundary of the "channel", bounded by the "upper" and "lower" curve determined by curve_fit. This function takes inputs of the locations of all sensors, loc_1 and loc_2, and outputs the list of sensors that can be eliminated by flying from loc_1 to loc_2.

(h) **dist_pt_sets and dist_PtoS**: This utility functions utilizes the concept of degrees of vertices to determine the distance between two sets and the distance between a point and a set, respectively. Between two sets of "semi-squares", or a point and a set, we would like to find the points from each side to connect. However, the points to be connected couldn't be "in-between vertices," as we want to maintain the completeness of a path in the square. Thus, this function looks for vertices in sets with degree 1, and eliminate those with degrees 2. Then, this function returns the distance given the above constraint.

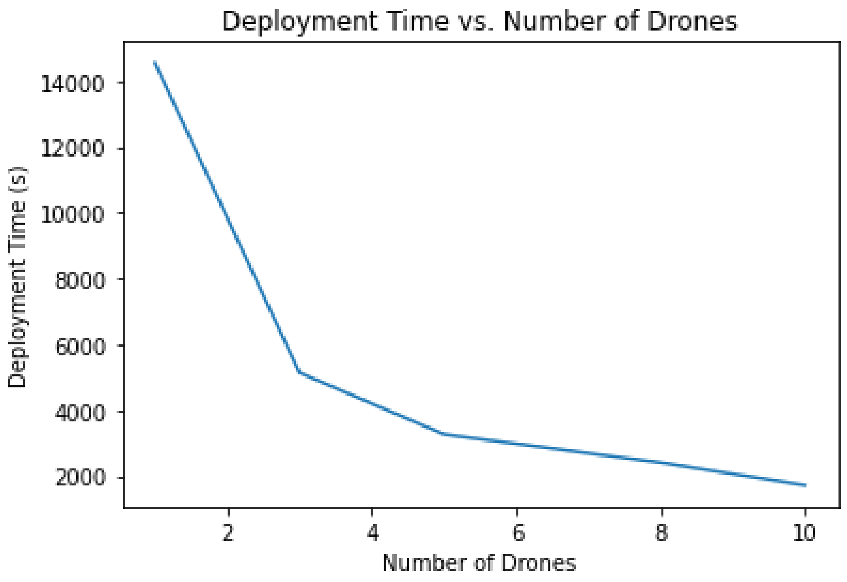
(i) **Miscellaneous**: Some other helper functions include the map_colorer, which colors the line of coordinates in between the two specified coordinates for trajectory display purposes, and lists_of_similar_weight, which distributes the time required for each small cycle to M drones, so that each drone takes approximately the same time to finish the cycles it got assigned. Another useful utility function that was used multiple times is fit_line, which is used with curve_fit to find the sensors covered by some drone path, and with map_colorer to visualize the trajectory.

3 Figures and Results

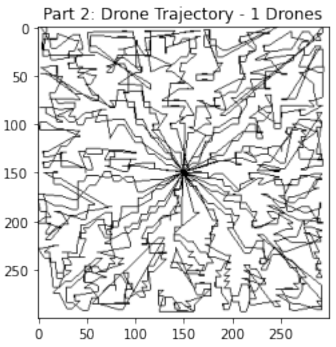
(a) **Scenario 1 - Total Distance & Trajectory With One Large Drone**: Our algorithm leads to a trajectory length of 13339.51427147393, which is reasonable, considering that our sensors are well-allocated, and the distribution has a gird-like feature. With our allocation, which is used class-wide, any distance that is in the range of 10-11k is deemed spurious, since our allocation algorithm from part 1 of this project guarantees that most sensors would be 9 units apart, with the exception when the new sensor falls through all three coverage thresholds. Thus, even with a semi-brute-force traversal algorithm, the total distance would be close to 300x300/6 = 15000. Observe in the trajectory plot that only a few segments have longer lengths, which coincide with the behavior of our edge connecting algorithm, that only towards the end will the insignificant divergent behaviors occur. Below is the trajectory of the drone:

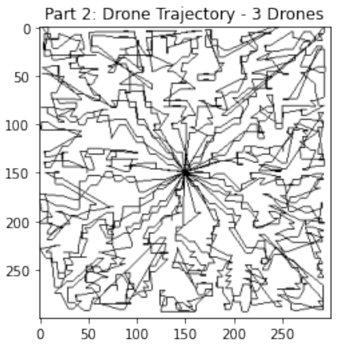


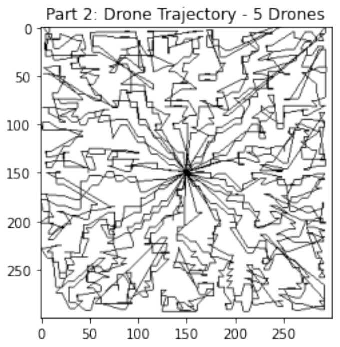
(b) **Scenario 2**: With multiple drones, our optimal deployment ratio (min [deployment time/number of drones]) comes out to be 172.3517072533582. Below is a figure plotting the number of drones vs the total deployment time for number of drones in [1,3,5,8,10].

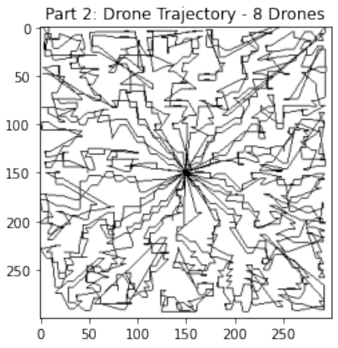


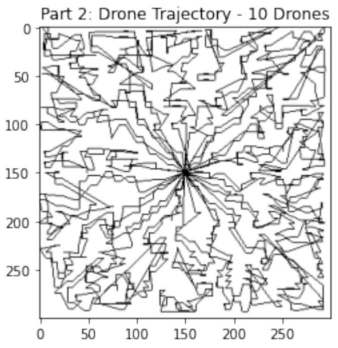
(c) **Scenario 2 - Trajectory for Each Number of Drones Deployed**: Below are the five figures associated with each number of drones used. Observe that because of the distance constraint imposed on the drones, all five plots have the exact same trajectory - our partitioning algorithm is designed such that each cycle that starts and end with the sink node would lie within the fuel and distance constraint. Since the polar-coordinate-based partitioning and the edge-connecting algorithm are extremely discreet, which don't involve randomness, the trajectory plots are identical. Thus, each "petal" in the radial-shaped plots is a cycle for each drone to complete, and for a multi-drone system, these "petals" are distributed among the drones to achieve optimal time. When the number of drones deployed surpasses the effects of the distance constraint (Ex. 20 drones), the number of "petals" would increase accordingly. However, with under 10 drones, the trajectories would look exactly the same.


Part 2: Drone Trajectory - 1 Drones


Part 2: Drone Trajectory - 3 Drones


Part 2: Drone Trajectory - 5 Drones


Part 2: Drone Trajectory - 8 Drones


Part 2: Drone Trajectory - 10 Drones