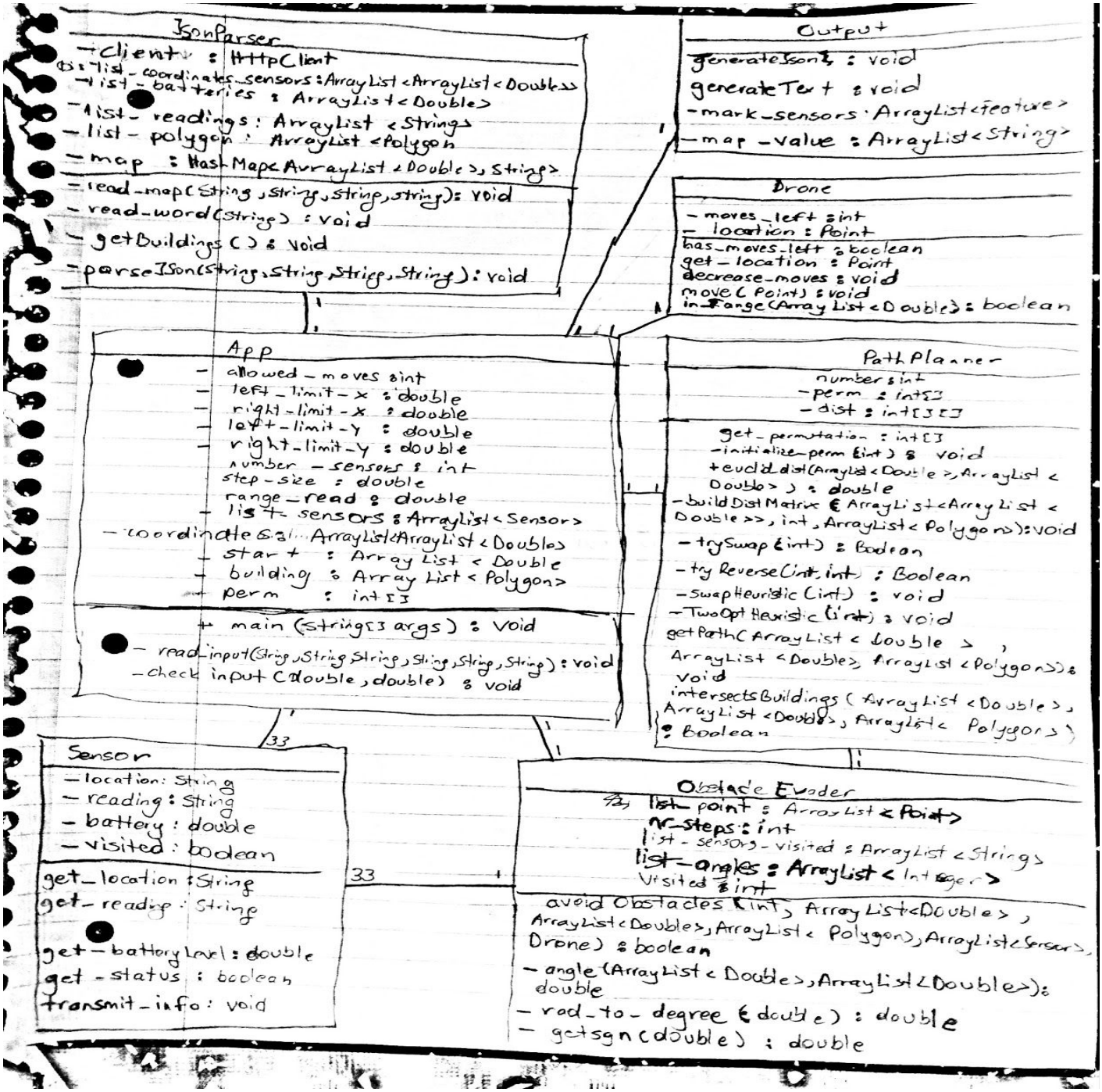
INFORMATICS LARGE PRACTICAL IMPLEMENTATION REPORT

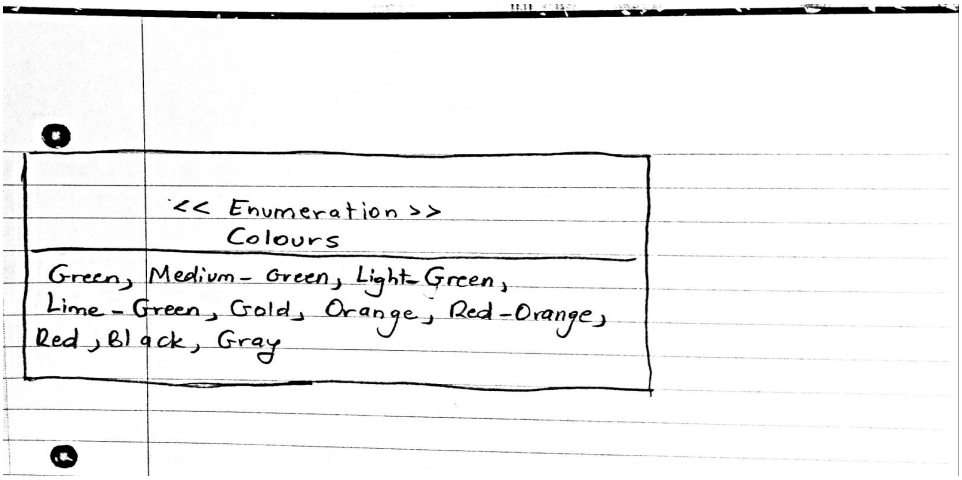
S1813106

1.Software Architecture Description

In this section I will provide a description of the overall software architecture of my application. I will explain why I identified the Java classes I've chosen as being the right ones for my implementation of Air quality Maps Drone. The UML Diagram in the following subsection will show the hierarchical relationships between the classes and provide a brief glance of the methods and variables present in each class as well as their respective visibilities.

1.1 UML Diagram





I did not have space for this enum in the above UML so I drew in another page. Nonetheless, It is part of the software architecture. Also, as I did not have any more space, I did not specify inputs for the methods in the Output class. They will be given in the class documentation part of the report.

1.2 Reasoning behind choosing these classes

-**App** : This is the main class of the project. It is where all of the features of the drone merge together to produce the results needed for this project. The app class uses JSon Parser to parse the data from the web server in a given day,starting location ,and port . It does not take seed numbers into account as I am not using it. Then it uses PathPlanner class to generate an optimal path and ObstacleEvader class to evade obstacles along the way.It stores all the data about sensors in an ArrayList<Sensor> , removing the need to have another class Map that would just store data collected by sensor. It also uses the Output class to generate the output. It mimics the behaviour of a drone controller since the drone has multiple independent parts that can be synergized together within the App class.

-**PathPlanner**: This class does not depend on any of the classes and is used to generate a list of sensors visited such that the distance from origin to all of sensors and coming back to origin is minimized. It uses SwapHeuristics and TwoOptHeuristics which I am going to talk about later.

-**ObstacleEvader**: This class is needed as the implementation for PathPlanner does not take into account that angles are multiples of 10 and that each step is 0.0003 in degrees. This could lead to the drone ending up in no-fly zones if only PathPlanner was to be used. Obstacle evader provides optional routes to the destination previously set by PathPlanner.

-**Output:** This class uses all of the data we have previously collected and generates the markers,line of path and also the text file. It uses the data from App to generate the output

-**JsonParser:** This class gets the project input from App class and retrieves the list of sensors and no-fly zones for a particular day which is defined by the input. It’s output is used by App class for the other tasks of the project.

-**Drone :** Drone class is used to hold information about the drone's current location and number of moves that drone has before it runs out of battery.

-**Sensor:** This class holds information about the sensor's location in what3words,battery,reading, and also if it is visited or not. It also has one method which sends info to the drone and from there we store the information about the sensor in an ArrayList<Sensor>, which has information about every sensor.

-**Colours:** They are used in the Output class to map colour names to their colour code. This makes my code more readable and more maintainable.

1.3 Class relationships

Generally speaking, I have loose coupling in my project as classes do not depend much on each other. This can ease project maintainability as one error in one of the classes will not cause error in all of the classes of the project. Also, it will help the developers isolate the errors.

Running the App class will instantiate a drone and 33 sensor objects which are put into an ArrayList<Sensor>. Sensor objects are also used in the ObstacleEvader Class and also in the Output Class. In both of these cases, they are given as input and are only instantiated in the App class. Also, an instance of the enum is used in the Output class to map colour codes to colour names.

Class Documentation

In this section I will discuss the classes in a more in-depth manner, specifying attributes within each class, the input and output of each method, their visibility and why they are integral to generating the movement of the drones.

2.1 App.java

This is the main controller class of the project. This class has the following **attributes:**

-allowed moves : this is of type integer and is final and default access. I use this attribute in drone class to initiate a drone with that number of allowed moves.

-left_limit_x: this is of type integer and is final and private. It is used in app class to check if a point is within the confinement area.

-right_limit_x: this is of type integer and is final and private. It is used in app class to check if a point is within the confinement area.

-left_limit_y: this is of type integer and is final and private. It is used in app class to check if a point is within the confinement area.

-righ_limit_y: this is of type integer and is final and private. It is used in app class to check if a point is within the confinement area.

-number of sensors: this is of type integer and is final and default access. It is used throughout the package and is final as I do not want to change the number of sensors.
-step_size: this is of type double and is private and final.I do not want to change step size
-range_read: this is of type double and is private and final. It is used to see if a drone is within range of reading a sensor
-list_sensors : this is of type ArrayList<Sensor> and is final and private. It holds the data from each sensor we read.
-coordinates: this is of type ArrayList<ArrayList<Double>> and is final and private. It holds information about the coordinates of all sensors.
-start: this is of type ArrayList<Double> and it is final and private. It holds the starting location.
-buildings: this is of type ArrayList<Polygon> and is final and private. It holds information about the no-fly zones.
- perm: this is of type int[] and is final and private. It will hold the order in which we will visit the sensors, calculated in the PathPlanner Class

The app class has the following **methods**:
-check_input: this method is private and is of return type void. It checks if the starting location is within the confinement area. If not, we exit the application.

-read_input: this method is private and is of return type void.It's inputs are day,month,year,port,and starting location in the form of the longitude and latitude. The inputs are given in a string format. It gets all the inputs of the project and it initializes and assigns list_sensors, no fly zones, the order in which we will visit the sensors and the coordinates of the sensors.It uses JsonParser parseJson method to get coordinates of sensors and no-fly zones. To get the order of visit, it makes use of PathPlanner getPath() and Path Planner get_permutation.It also uses check_input method to check if starting location is valid

-main(String[] args) : this is where the application is launched. First, the application reads the input through read_input and assigns the appropriate values to attributes as defined above. If the starting location is invalid, we exit the program. Then, it creates a new Drone with arguments starting location and allowed moves. Then, it tries to go to the specified order of sensors in perm avoiding obstacles along the way using the ObstacleEvader class avoidObstacles method. If at any point the drone is out of moves we exit the loop of trying to go to all sensors. Then output is generated using the Output class generateJson and generateText methods. If the drone was successful we print Success. Otherwise we notify the user that the task was incomplete.

2.2: **PathPlanner**
This class generates an optimal path that starts from the starting location,goes through each of the sensors once and then returns back to the original location.This class has the following **attributes**:
-number : is of type integer and is final and default access. It used to denote number of sensors
-dist: is of type double[][] and is private. dist[i][j] is used to measure distance from point i to point j.
-perm: is of type int[] and is private. It is used to store current order of visiting nodes.

PathPlanner class has the following **methods**:
-get_permutation: returns a int[] and has default access so that it can be used throughout the project. It is used as a getter for attribute perm.

-initialize_perm: it does not return anything but initializes the perm array. It is private and gets an integer as an input denoting number of nodes
-buildDistMatrix: it does not return anything but it builds the dist[][] attribute. It is private and it gets as input the list of coordinates of the sensors, number of points to go through and also the no-fly zones.

-trySwap: it is private and returns a boolean if swap is worth it. It gets the node number as input and returns true if swap is worth it.

-swapHeuristics: it is of return type void and gets the number of nodes as an input. It is private and makes use of the trySwap method.

-tryReverse: it is private and returns a boolean if reversing is worth it. It gets as an input 2 integers denoting which sequence to reverse and returns true if reversing the sequence will give better results.

-TwoOptHeuristics: it is of return type void and gets the number of nodes as an input. It is private and makes use of the trySwap method.

-intersectsBuildings: it is of return type boolean and gets as input the coordinates of a point in ArrayList<Double>, coordinates of another point in ArrayList<Double> , and an arraylist of polygon denoting the no-fly zones. It returns true if the line between the 2 points intersects any of the buildings. It has default access so it can be used in all the classes in the package.

-getPath: is of void return type and has default access. It gets an input list of coordinates of sensors, number of points and also the no-fly zones to generate the best path.
Please note that I am not explaining in detail the methods as It intersects with the algorithm part.

2.3: **ObstacleEvader**:
This class makes sure that any of the moves I want to make is not intersecting any of the buildings, and generates optional paths for the drone to reach the sensors

The class has the following **attributes**:
-list_point : is of type ArrayList<Point> and has default access. It stores information about the list of points on the drone's path.
-nr_steps : denotes current number of steps used by drone. It is used to print the amount of steps it took for the drone to finish the task. It is of type integer and has default access.

-list_sensors_visited: it is of type ArrayList<String> and has default access. It stores information about location of sensors visited after every move. If no sensor was visited, the application add “null”.
-list_angles: it is of type ArrayList<Integer> and has default access. It stores information about angle used after every step.
- visited: it is of type visited and has default access. It stores information about the number of sensors visited.

The class has the following **methods**:
-avoidObstacles: has return type boolean and gets as input order of visit as int, starting point as ArrayList<Double>, destination as ArrayList<Double>, no-fly zones as ArrayList<Polygon> ,and list of sensors as ArrayList<Sensor>. It tries to get to its destination without crossing any of the no-fly zones. Mone on this on the algorithm part.

-angle: it is private and it returns a double. It gets as input a point in ArrayList<Double> format and a point in ArrayList<Double> format and finds its angles in degrees.

-rad_to_degree: it is private and returns a double. It takes as an input a double in radians and converts it in degrees.

-getsgn: it is private and returns an int. It takes as input a double and returns 1 if double is bigger than 0 or -1 if double is smaller than 0.

2.4: Output
This class generates Json and text files required by the assignment
This class does not have any **attributes**.
Output class has the following **methods**:
-generateJson: it is of return type void and has default access. It takes as input list of sensors as ArrayList<Sensor> , coordinates of sensors as ArrayList<ArrayList<Double>>, list of points done by the drone as ArrayList<String>,day as String, month as String,year as String , and also order of sensors visited as int[]. It makes use of the mark_sensors method to mark the sensors on the map and then it also writes the path of the drone saving it as a geojson file.

-mark_sensors: it is private and returns an ArrayList<Feature>. It takes an input list of sensors as ArrayList<Sensor> , coordinates of sensors as ArrayList<ArrayList<Double>> and order of visiting sensors as int[]. It makes use of the map_value method which maps sensor readings to the features each sensor has.In addition to that, it prints that sensors needs repairing if battery is low.

-map_value: it is private and returns an ArrayList<String>. It takes as input sensor reading as double, if sensor is visited as boolean and if sensor has battery as boolean. It makes use of Colours Enumeration I have defined to map colour names to colour codes to make code more readable.

-generateText- it is of return type void and has default access. It takes as input list of location of sensors visited as ArrayList<String>, list of angles of moves as ArrayList<Integer>, list of points visited by drone as ArrayList<Point>, day as String, month as String, and year as String. It generates the text file as specified in the assignment specification.

2.5: JsonParser

This class is responsible for parsing data from the web server.
This class has the following **attributes**:
-client: is of type HttpClient and is private and final. It is used to process the requests to the web-server.
-list_coordinates_sensors:it is of type ArrayList<ArrayList<Double>> and it holds information about coordinates of the sensors. It is private
-list_batteries: it is of type ArrayList<Double> and it holds information about the battery level of each sensor. It is private
-list_readings: it is of type ArrayList<String> and it holds information about the readings of each of the sensors. It is private
-list_polygon: it is of type ArrayList<Polygon> and it holds the no-fly zones. It is private
-map : it is of type HashMap<ArrayList<Double>,String> and it maps sensor coordinates to its corresponding what3words location. It is private
This class has the following **nested classes**:
-Map: it is used to create json records of map readings.
-what3Words: it is used to create json records of word readings
JsonParser class has the following **methods**:
-get_coordinates: getter for list_coordinates_sensors attribute
-get_battery: getter for list_batteries attribute
-get_readings: getter for list_readings attribute
-get_buildings: getter for list of polygon attribute
-get_mappings: getter for map attribute

-readMap: is of return type void and is private. It gets as input String day, String month, String year and String port and makes a request to get the data. It uses readWord method to get location of each sensor and the fills list_readings,list_batteries,list_coordinates_sensors,and list of maps attributes accordingly.

-readWord: it takes as input the path to the word as a string and it generates the coordinate of that sensor in the format of ArrayList<Double>. It is private and of return type void.

getBuildings: it is private and of return type void. It gets port number as an input in a String and get the no-fly zones and assigns them to list_polygon attribute

parseJSon: it has default access and is of return type void. I want to use it in other classes in my package so that is why it has default access. It gets as input day,month,year, and port separately

as a String and it brings together readMap and getBuildings to get all of the readings in one method.

2.6: Sensor

It is used to have a sensor object within the project for added functionality.
It has the following **attributes**:
-location: of type String and private. It is the location in what3words
-reading: of type String and private. It is the pollution reading for that sensor
-battery: of type double and private. It is the battery reading of the sensor
-visited: of type boolean and private. It denotes if sensor has been read or not
Sensor class has the following **methods**:
get_location: is a getter for location attribute
get_reading: is a getter for reading attribute
get_batteryLevel: is a getter for battery attribute
get_status: is a getter for visited attribute
transmit_info: is a method of type void that sets the visited to true. This allows the application to access the sensor readings and mark them on the map.

2.7: Drone

Drone object has the following **attributes**:
moves_left: it is of type int and is private. It holds the information about the number of moves the drone has before it runs out of battery
location: it is of type Point and is private. It holds information about the drone current location.
Drone object has the following **methods**:
-has_moves_left: it returns a boolean indication if drone has any moves left. It has default access
-get_location: it is a getter for location attribute

-decrease_moves: it is of void return type and decreases the number of moves the drone can make by 1. It has default access

-move: It is of void return type and has default access. It takes as input a point and it moves the drone to that point

-in_range: it is of boolean return type and has default access. It takes as an input the coordinates of a point as ArrayList<Double> and returns true if drone is in range of reading that point

2.8: Colours

This is an Enumeration class holding information about colour codes of the colors defined by this assignment. We can access its code by Colour.get_color_code

3.Drone Control Algorithm

The main algorithms that I wrote for drone control are in the PathPlanner and in the ObstacleAvoider classes. PathPlanner class is used to determine the order in which the drone will visit the sensors without accounting for angles being multiples of 10 or that moves are 0.0003 degrees. It will generate an optimal path and the application will try to follow that path and not go to a no-fly zone using the ObstacleEvader class.

3.1 PathPlanner

The way I decided to solve the problem is by first defining the distance between each of the nodes. I define nodes such that each sensor is assigned a node number from 1 to 33 and 0 being the starting location. So fundamentally, I was trying to solve a Travelling Salesman Problem starting from node 0 and ending at node 0, visiting all the nodes along the way. I used euclidean distance as the measure of defining distance between nodes.

To account that a path may intersect with one of the buildings, I wrote the intersectBuildings function. So if a path between 2 nodes intersects one of the no-fly zones I assign dist[i][j] a value of 1000. This is too big for our map so the algorithm will try to avoid the path between nodes that intersect with buildings.

Next, after having the distances between nodes, I need to have an algorithm to get the smallest weighted graph that starts at node 0 and ends at node 0 . For my implementation I used swap Heuristics together with TwoOptHeuristics.

In Swap Heuristic , we explore the effect of repeatedly swapping the order in which a pair of adjacent nodes are visited, as long as this swap improves (reduces) the cost of the overall cost.The method which governs the high-level control of this heuristic (the repeated swapping, until we can see no improvement from these swaps) is swapHeuristic. It makes use of trySwap method to swap every node until i real a local minimum. This method is not that efficient so I needed to use it and another method to get best results. That is TwoOptHeuristic.

The Two-Opt Heuristic is another heuristic which repeatedly makes “local adjustments” until there is no improvement from doing these. In this case the alterations are more significant than the swaps - the method `TwoOptHeuristic` repeatedly nominates a contiguous sequence of nodes on the current tour, and proposes that these be visited in the reverse order,if that would reduce the overall cost of the tour. It makes use of tryReverse and finds a local minimum. I would get results of not more than 100 steps when i used both swap Heuristic and Two-Opt-Heuristic.
But, I was not yet accounting for angles and the drone making a move before reading a sensor.That is why ObstacleEvader class was needed.

3.2 ObstacleEvader

In obstacleEvader, I define the point where the drone is on the map. Main method here is avoidObstacles. First I check if the drone is out of moves, I break out of the loop. Otherwise, I continue. I also have a special case that if the drone has visited all the sensors and it is close to the starting point, the drone can stop moving and I print “success”.

Otherwise, I find the angle between the current point and the destination that is set beforehand in the main class. This destination is just the location of the node number to be visited next as defined by the PathPlanner. I find the angle using the angle() method and I round it so that is a multiple of ten. Then I ask the drone to move in that angle.
If the move is not intersecting any of the polygons, I execute it and I add the new location to the list of points visited by the drone. I check that the line between current location and next move does not intersect any of the polygons using the PathPlanner intersectBuildings method.

If the move actually intersects the buildings I try to find an optional route for the drone. I first define where the point is with respect to the drone’s location and try to move with a step size in a direction positive to it. What I mean by this is that if the destination is up and right with respect to the drone’s location, I check if I can move up or right using the same method as defined above.
If all of these moves intersect the buildings i try to move in a negative direction with respect to destination location. In my previous example that would be moving left or down. I check if these moves intersect the buildings and then I move. Since the buildings are more than 0.0003 apart, one of these moves will actually get to a location that is not on the polygon.

Then, I add the angle to the list of angles with which the drone moved. Then I check if the drone is within range of the sensor. If it is, the drone reads the sensors and the transmit_info() method is invoked causing the information to be saved in the app class.

If no sensor is within range I repeat the process until I reach the sensor or drone is out of battery. If the task was successful I print that it was a success. Otherwise, I print”Task was incomplete”. In the output class I also print the location of the sensors that need battery replacement to notify the users that some sensors need battery replacement.

3.3 Algorithm run on 2 dates

The following figures show flight path of drone on 2 different maps:

Date: 04/04/2020

Date: 11/11/2020

3.4 Testing

To test my algorithm I wrote a script to run the algorithm to get map data from 2020 and 2021. The script's name is **run_tests.py**. It completed the task in 354 out of 366 days in 2020 and in 353 out of 365 in 2021 getting a success rate of nearly 97%. In those successes, the drone reads the sensor and comes back in an average of 100 steps getting good results. There is still room for improvement as it fails 3% of tests.

References:

https://en.wikipedia.org/wiki/Travelling_salesman_problem
https://en.wikipedia.org/wiki/2-opt