Informatics Large Practical

Coursework 2 Report

Justin Howe

December 10, 2020

Contents

Chapter 1

Software Architecture Description

1.1 Overview

This section explains the reasoning behind for each class, rather than the semantics.

1.2 App

This main class is ultimately responsible for running the drone application simulation, which would require calls from other classes' methods especially from AppUtils.java to fetch data for sensors. In addition, this class would be required generate the GeoJSON map and flight path text files into the current working directory. Finally, useful information about the drone application simulation is printed into the console for researchers and developers working on the project possibly in the future.

1.3 AppUtils

The application utility class is needed to handle the back-end networking of the web application. In particular it is used to access web servers and its content files and folders. It is responsible for creating the HTTP Client, sending a HTTP request, and receiving a HTTP Response. Given the response, this class provides methods for parsing the JSON and GeoJSON files found in the web server content.

1.4 Direction

The Direction class is created to determine which direction the drone should travel towards. It would attempt to move towards its desired sensor and take the reading. As described in the Coursework requirements documentation, the scenario accurately portrays the constraints where the drone can only fly in multiples of 10 inclusively between 0 to 350 degrees.

1.5 Drone

The Drone class contains the main methods for the drone's greedy flight path algorithm. As described in the coursework document, it is responsible for moving and taking reading of the sensors in order as explained in the drone's lifecycle.

Since the drone's flight path algorithm is greedy, there may exist a possibility where the drone would be stuck near the No Fly Zones as it would move back and forth until it runs out of moves. Therefore, the class also provides the algorithm to attempt the drone to get out of that loop.

If the drone contains sufficient number of moves after collecting all the sensor point readings, it will then proceed to return back to the original position as close as possible, provided the total number of moves is less than or equal to 150.

1.6 DroneUtils

The Drone helper class would be useful for defining the drone's helper methods for refactoring and enhanced readability purposes. For example the drone itself shouldn't calculate the distance between one point to another or create a String representation of its movement, so helper methods from these scenarios should be classified under such a helper or utility class.

1.7 Map

The Map class is responsible for creating the visuals of the drone flight path and the 33 colorized sensors, each assigned with a marker symbol, in the GeoJSON map. The class also checks if the drone's anticipated movement path would intersect or collide into any of the No-Fly-Zones buildings, as the drone is not allowed to travel through the No-Fly-Zones. In addition, it contains helper methods which maps a reading value to a RGB string attribute and a marker symbol. Helper methods also include creating the drone anticipated movement path and the Polygons using Path2D and Line2D.

1.8 NoFlyZoneBuilding

While the Polygon class from the MapBox Java SDK may seem more ideal for implementing the No-Fly-Zones buildings, but this class was created to clearly identify these No-Fly-Zones. In the future, if required, this class could also be further expanded to implement additional features in the map. However, for the current time being, this class contains the coordinates and names for each building.

1.9 Position

The position class which determines the longitude and latitude for the drone and sensor points in the map. It also contains the method to calculate the drone's next anticipated position, given the direction.

1.10 SensorPoint

The SensorPoint class is needed to obtain details of each sensor's location, position, battery percentage and the sensor reading value.

Chapter 2

Class Documentation

2.1 App

The main class, which runs the drone application simulation, should be kept simple but efficient.

2.1.1 Method: public static void main(String[] args) throws Exception

The application first checks if there are 7 arguments, each in the form of a String, are executed from the command line. From those arguments, the list of sensors and No-Fly-Zone buildings in a date-specified map would be obtained. The GeoJSON Map would be generated in the *Map.generateMapGeoJson(List<SensorPoint> sensorPoints)*.

The drone is first created or instantiated with the starting position and then the drone flies using a greedy algorithm, where the drone would have to take readings for the list of sensors and additionally avoid the No-Fly-Zone buildings. The algorithm will be further explained in the Drone class.

A final FeatureCollection would be generated, showing the drone's flight path and the map showing the sensors. Finally, the flight path text files and GeoJSON readings for the particular day, month, and year are created. In addition, useful printed outputs for researchers and developers include:

1. Date of Map Accessed
2. If the drone taken readings for all sensors in the map
3. Did the drone return close to its original position?
4. Number of Moves Taken
5. Execution time duration

2.2 AppUtils

2.2.1 Methods:

All of the following methods are static in this class because an object of Map does not need to be created but methods should be referenced by that class. The helper methods are private, which are used only within the class and for encapsulation purposes which prevent other classes from calling those methods. The public main utility functions are called in the main App.java class which requires the SensorPoints' data, including location and the NoFlyZoneBuildings for the drone to avoid.

1. **private static String createURLStringForBuildings(String portStr)**:
   Taking the WebServer port number as a String, this method creates the URL String for obtaining the coordinates and names for each No-Fly-Zone building.
2. **private static String createURLStringForSensorPointsAirQualityData (String dayStr, String monthStr, String yearStr, String portStr)**:
   The day, month, year together with the WebServer's port number in the form of a String are obtained

from the command-line arguments. This method creates the URL String for obtaining the date-specified map which contains the 33 SensorPoints information, where each contains the What3Words location, percentage battery charge and the sensor reading value.

3. **private static String createURLStringForSensorPointLocationDetails (String portStr, String sensorPointLocation)**:
   From the SensorPoint's What3Words location, it creates the URL String for obtaining the coordinates (which includes the longitude and latitude) for that particular location.

4. **private static HttpRequest createHttpRequest(String urlString)**:
   Creates the HttpRequest using try-catch block using the URL given as a String format.

5. **private static HttpResponse<String> sendHttpResponse(HttpClient client, HttpRequest request)**:
   Given the HttpClient and HttpRequest, this method attempts to collect the response after the HttpRequest is sent by the HttpClient using a try-catch block.

6. **private static BufferedReader readFile(URL url)**:
   This helper function reads the file using InputStreamReader, and is used for parsing the geojson and json files in the WebServer.

7. **private static List<NoFlyZoneBuilding> parseGeoJsonBuildings(URL url)**:
   Given the HttpResponse URL, the function parses (or deserializes) the no-fly-zones.geojson file, which contains information for the NoFlyZoneBuildings found in the WebServer which is found in the buildings top-level folder. Returns the list of NoFlyZoneBuildings which contains the coordinates.

8. **private static double[] parseJsonSensorPointLocation(URL url) throws Exception**:
   Given the HttpResponse URL, this function parses (or deserializes) the sensor's location which corresponds to a What3Words address, where the top-level folder is words. It obtains the "coordinates" which contains the longitude and latitude for each sensor.

9. **private static List<SensorPoint> parseJsonAirQualityData(String portStr, URL url) throws Exception**:
   Given the HttpResponse URL, this method deserializes the air-quality-data.json file into a list of SensorPoints. Each contains the coordinates (longitude and latitude), battery charge percentage and the sensor's reading.

10. **public static double[] fetchSensorPointLocationCoords(String portStr, String sensorPointLocation) throws Exception**:
    The main function first attempts to access the WebServer by creating a HttpClient and HttpRequest. A response would be received from *AppUtils.sendHttpResponse(client, request)* Using helper method *AppUtils.parseJsonSensorPointLocation(String portStr, URL url)*, the sensor point coordinates (which includes the longitude and latitude) are found. Catches any Exception would can be found in the parsing function.

11. **public static List<SensorPoint> fetchSensorPointData(String dayStr, String monthStr, String year, String portStr) throws Exception**:
    The main function first attempts to access the WebServer by creating a HttpClient and HttpRequest. A response would be received from *AppUtils.sendHttpResponse(client, request)* Using helper method *AppUtils.parseJsonAirQualityData(String portStr, URL url)*, the sensor point information (including the percentage battery charge and reading value) can be found. Catches any Exception would can be found in the parsing function.

12. **public static List<NoFlyZoneBuilding> fetchBuildingCoordinates(String portStr) throws Exception**:
    The main function first attempts to access the WebServer by creating a HttpClient and HttpRequest. A response would be received from *AppUtils.sendHttpResponse(client, request)* Using helper method *AppUtils.parseGeoJsonBuildings(String portStr, URL url)*, the NoFlyZoneBuildings' coordinates (which includes the longitude and latitude) are found. Catches any Exception would can be found in the parsing function.

2.3 Direction

While there are 35 possible direction angles and it is possible to list them all manually, but this approach would be rather time-consuming and inefficient. Therefore, a class is needed to obtain the degrees the drone would travel in.

2.3.1 Variables

degrees- integer which shows the angle degree value at which the drone can fly in

2.3.2 Constructor

If the angle degrees value is a negative value or greater than 350, or if the angle degrees value is not a multiple of 10, then an IllegalArgumentException is thrown because it would not satisfy the constraints as described in the requirements document. If the constraints are satisfied, then the Direction class can be created to assign the angle degree values.

2.3.3 Method:

public int getDirectionInDegrees(): Returns the direction angle value in degrees

2.4 Drone

The Drone class is responsible for running the greedy algorithm where it follows the order of moving and taking readings of sensors before returning close to its original position. It also handles the situation if the drone gets stuck moving back and forth. More details of the drone's algorithm can be referred at page 11.

2.4.1 Variables

Following instance variables *List<SensorPoint> visited* and *List<SensorPoint> notVisited* are static because the DroneUtils, the Drone helper class, uses getter methods to get the list of sensors visited and not visited.

1. **private Position position**: Shows drone's current position
2. **private SensorPoint nextSensorPoint**: The sensor which the drone anticipates to move towards and take readings of
3. **private List<Position> travelledPath**: The list of Positions the drone has been at
4. **private static List<SensorPoint> visited**: The list of sensors the drone has taken readings of
5. **private static List<SensorPoint> notVisited**: The list of sensors the drone has not taken readings of
6. **private List<NoFlyZoneBuilding> buildingsToAvoid**: The list of No-Fly-Zones buildings
7. **private List<String> movements**: String text representation for movements- this attribute is used to generate the drone's flight path in the format as a text file in App.java
8. **private boolean sensoredAllPointsAndNearOriginalLocation**: Attribute which checks if the drone has sensored all points and has returned to close to its starting position
9. **private int lastBestDirectionAngle**: The previous direction the drone flied in
10. **private int numberOfMovesRemaining**: The drone's remaining number of moves
11. **private boolean isStuck**: Attribute if drone is stuck where the drone moves back and forth

2.4.2 Constructor

public Drone(Position position, List<SensorPoint> points, List<NoFlyZoneBuilding> buildingsToAvoid):
When instantiated, the drone would be created with the starting position which contains its starting longitude and latitude. The drone would have the list of SensorPoints which are not visited and knows which buildings to avoid. The drone also respects the lifecycle pattern which first makes a move and then takes a sensor reading if in range which is within 0.0002 degrees of that particular sensor. The drone would start with 150 moves remaining.

2.4.3 Methods:

1. **public void generateGreedyFlightPath()**:
   This is the main method where the drone attempts to fly, take readings for all sensors and return close to its original location, assuming if the drone completes its lifecycle within 150 moves. It also calls out the method to handle if the drone is stuck where it moves back and forth. More information regarding my particular algorithm can be found in page 11.

2. **public void move()**: Each call of this function represents one unit of movement. After knowing which sensor point the drone should fly towards and take readings of that sensor, this method searches and determines which direction the drone should fly in if the drone is not stuck.

3. **public String takeReading(Position dronePosition)**: This method is a String which returns the SensorPoint. By default, the next sensor point (nearest sensor point) is not read yet so the value is assigned as null.

   From the drone's current position taken as an argument in this function, the reading can be taken if the distance between the drone's position and the next sensor point is within 0.0002 degrees. The reading would record and return the sensor's name. In addition, it cannot be called before the *move()* method, as this would contravene the drone lifecycle.

4. **public void returnTowardsOriginalLocation()**: After the drone has taken the readings of all sensor points, it attempts to return back to its original location as best as possible without colliding or moving into any of the NoFlyZones.

5. **public void handleStuckError()** As mentioned earlier, the drone may be stuck in a loop where it moves back and forth as it must avoid the No-Fly-Zones buildings. Used in *move()* and *returnTowardsOriginalLocation()*, the drone continously checks for both if it has moved back and forth twice to the same position as before. If so, *forceMove()* is called and *isStuck* attribute would be assigned to true, assuming if the number of moves remaining are sufficient.

6. **public void forceMove()**: The drone would attempt to move in the direction away from the building, where it usually gets stuck. I have manually determined the angles it could go at are in the order:

   $$100, 170, 180, 190, 260, 280, 350, 10, 180$$

   The reason why these angles were chosen can be referred in section 3.2. Once the angle is decided where it doesn't collude into any of the NoFlyZones, the drone then forced to move three times at that angle. In addition, the movements are recorded and exits the method.

   The attribute *isStuck* would then be set to false.

2.4.4 Getter and Setter Methods:

While the getter methods should be self-explainatory, the setter methods should be further explained. Take note the methods *getVisitedSensorPoints()* and *getNotVisitedSensorPoints()* are static.

1. **public void setPosition(Position newPosition)**: Sets drone's current position with the new position given as the argument.

2. **public void setNextSensorPoint(SensorPoint nextSensorPoint)**: Sets the drone's targeted sensor with the new sensor point the drone would like to visit.

3. **public void addPositionForTravelPath(Position position)**: Adds the drone's position to its travelled path

4. **public void setAngle(int angle)**: Sets the last direction angle the drone has travelled with the given angle.

5. **public void setReturned()**: The attribute *sensoredAllPointsAndNearOriginalLocation* is assigned as true. This is critically used to indicate the drone's algorithm is completed when it has visited all sensors and is close to its starting position (within 0.0002 degrees).

2.5 DroneUtils

Helper class for the drone. The DroneUtils should not be instanitated as an object to reference its methods. However, the class should be called to reference its methods. Therefore its methods should be static.

2.5.1 Methods

1. **public SensorPoint findClosestNotVisitedSensorPoint (Position dronePosition)**:
Obtains the list of all the air-quality sensors the drone has not taken readings of yet, and then finds the closest sensor point based from the Euclidean distance heuristic.

2. **public static double calculateDistance (Position p1, Position p2)**: This helper function is used to calculate the distance between the drone's current position and its next anticipated position or the distance between the drone's current postiion to a sensor point's position respectively using the Euclidean distance formula.

3. **public String createStringMovement(int moveNumber, Position droneCurrentPosition, int lastBestDirectionAngle, Position droneNewPosition, String sensorPointStrFormat)**:
Helper function which creates a String representation of each drone's movement. This includes when the drone returns back close to its original position.

   The format for each movement line follows as: *int,double,double,int,double,double,String* as described in the flightpath-DD-MM-YYYY.txt file.

4. **public boolean hasVisited(SensorPoint point)**: Checks if the given SensorPoint's reading has been recorded from the list of visited sensors.

5. **public boolean meetsAllRequiredConstraints (Position droneNextPosition, Line2D.Double line, Path2D.Double building1, Path2D.Double building2, Path2D.Double building3, Path2D.Double building4)**:
First this method checks if the drone's next anticipated position is within the confinement area. In addition, this function also checks if the drone's anticipated drone paths would intersect any of the buildings.

2.6 Map

The Map class is responsible for generating the GeoJSON map and creating the visuals of the drone flight path and all NoFlyZoneBuilding boundaries.

2.6.1 Methods

All methods in Map class are static, because no instance of the Map class is created in the App main class.

1. **public static String readingValueToRGBString(String readingValueStr)**:
Given the sensor's air quality reading value as a String, it maps it to a RGB colour as a String as shown in Figure 5 of the coursework document. Values should first be checked if it is null or "NaN" whose values' attributes are a String before attempting to map the value as a double. This would be used to assign each sensor point a RGB colour.

2. **public static String readingValueMarkerSymbol(String readingValueStr)**:
   Maps the given air quality sensor reading to the appropriate marker symbol based from the coursework document in Figure 5. This can be simplified. First checks for null values, if so, we can assume the sensor reading cannot be read as the sensor battery level is too low. Therefore, the marker symbol is 'cross'. If the reading value is inclusively between 0 to 127, the marker symbol is 'lighthouse'. If the reading value is inclusively between 128 to 255, the marker symbol is 'danger'.

3. **public static String generateMapGeoJson (List<SensorPoint>sensorPoints)**:
   After retrieving the information of all SensorPoints from the WebServer, it then creates the features and adds properties for those SensorPoints, which ultimately creates the GeoJSON format for the Map to be displayed. The polygon showcasing the boundaries for NoFlyZoneBuildings do not need to be shown.

4. **public static Line2D.Double createLine2D(Position p1, Position p2)**:
   Given the two positions, this method to create the anticipated drone's anticipated linear flight path.

5. **public static Path2D.Double createPath2D(NoFlyZoneBuilding building)**:
   Used to create the polygon representation for all of the No-Fly-Zone buildings.

6. **public static boolean intersects (Path2D.Double path, Line2D.Double line)**:
   Given the No-Fly-Zone building polygon as a Path2D and all a drone flight path as a Line2D, this method is used to check if the drone's anticipated flight path would intersect any of the NoFlyZoneBuildings in the Map using the PathIterator class. It determines if any of the building segments intersect with the drone path at all [3].

7. **public static String generateFinalGeoJson(List<Position>path, FeatureCollection fc)**:
   Creates a GeoJSON String which generates drone's flight path from drone's designated travel path and FeatureCollection Map

2.7 NoFlyZoneBuilding

2.7.1 Variables:

Includes **private String name** and **private List<Point> coordinates**

2.7.2 Constructor:

Self explainatory: assigns the name and list of Point coordinates to the No-Fly-Zone building.

2.7.3 Methods:

Only includes **getName()** and **List<Point> getCoordinates()** which are both getter methods.

2.8 Position

As explained earlier, it is a useful class which determines both the drone and any SensorPoint's latitude and longitude. In addition, it can also calculate the next anticipated position of the drone knowing the direction.

2.8.1 Variables

1. **private double longitude**: Displays the longitude of the SensorPoint or drone
2. **private double latitude**: Displays the latitude of the SensorPoint or drone
3. **private final double droneMovementDegrees = 0.0003**: This instance variable is final, because the drone can only move 0.0003 degrees per move, no more or less.

2.8.2 Constructor

The constructor is created for the drone, when its starting longitude and latitude are initalized.

2.8.3 Methods

1. **public boolean isWithinConfinementArea()**: Checks if the drone's position is within confinement area as required by the document.
2. **public Position nextPosition(Direction direction)**: Given the drone's anticipated direction, this method calculates the drone's new position.
3. **public double getLongitude()**: Getter method for obtaining the longitude value
4. **public double getLatitude()**: Getter method for obtaining the latitude value

2.9 SensorPoint

SensorPoint contains the What3Words location, the sensor reading value as a String, the sensor's position (containing the latitude and longitude) and battery charge percentage.

2.9.1 Variables

1. **private String location**: Name of the What3Words encoded address location
2. **private String sensorReading**: The sensor reading should be first read as a String, because the reading may be a real number value between 0 to 255, or "null" or "NaN".
3. **private Position position**: the class variable is required to obtain both the longitude and latitude attributes for each SensorPoint, which are found in the Position class.
4. **private double batteryPercentage**: the double attribute is required to determine whether the sensorReading could even be read

2.9.2 Constructor

Self explainatory: obtains information for each sensor in the map. It would assign the location, sensor reading, the position, and battery charge percentage.

2.9.3 Methods

For Java encapsulation purposes, I only have public getter methods. This includes **public String getLocation()**, **public Position getPosition()**, **public double getBatteryPercentage()** and **public String getSensorReading()**.

Chapter 3

Drone Control Algorithm

3.1 Problem:

The drone's ultimate goal is to attempt to collect readings for all of the air quality sensors inside the given map within 150 moves. In addition, the drone must follow the life-cycle where each unit of movement is exactly 0.0003 degrees and may take a reading if the distance between the drone's current position and the sensor it is going towards to is less than 0.0002 degrees.

3.2 Strategy:

Located in the *generateGreedyFlightPath()* inside the Drone class, the algorithm is greedy. The drone can be in one of the three possible scenarios and I have explained the strategy for each scenario.
Assuming if the drone has sufficient number of moves remaining:

1. **The drone has not yet collected all of the air quality sensors' readings in the map.**
   Using a greedy approach, the drone finds the nearest air quality sensor out of all the sensors which are not visited yet by the drone. It would follow the drone lifecycle where it would move towards that closest sensor without colluding or flying into any of the NoFlyZone buildings and then take a reading if applicable. A brute-force algorithm is used to check for the direction angle the drone should move in. The drone checks for every possible direction angle in multiples of 10 where there are 35 different possible angles to choose. Afterwards, if applicable, take the reading of that sensor. This algorithm would continue until the drone has taken readings for all of the air quality sensors.

2. **The drone has collected all of the air quality sensors' readings but has yet to return close to its starting position.**
   Once the drone has visited all of the air quality sensors, the drone would attempt to return back to its starting position as close as possible without colluding or flying inside the NoFlyZone buildings. I manually determined if the drone's distance between the drone and its starting position is within 0.0002 degrees, the drone would be considered close to its starting position.

3. **The drone has collected all of the air quality sensors' readings and has return close to its starting position.** The algorithm can then be finally terminated.

The Possibility where the Drone gets Stuck: This possibility may occur in Scenario 1 or 2 above. I therefore have enhanced my greedy algorithm to take into consideration if the drone gets stuck where it moves back and forth twice to the same position where it is called in. The main idea is to get the drone move away from the building so it can continue its algorithm.

I have manually determined the angles the drone could possibly move in the direction order of:

$$100, 170, 180, 190, 260, 280, 350, 10, 80$$

Derived from the unit circle, the order of the list starts from the 2nd quadrant of a rectangular coordinate system, and goes counter-clockwise as this can be visualized in Figure 3.1.

This order is established because it usually gets stuck in Appleton Tower or Appleton Forum. Therefore, it is preferable for the drone to attempt to move away towards the left direction first.

From that list, the drone checks if that the drone moving in that direction would collide into any of the buildings. If not, the drone would force move three times in that direction, and those movements are recorded. This would then allow the drone to continue its greedy algorithm.

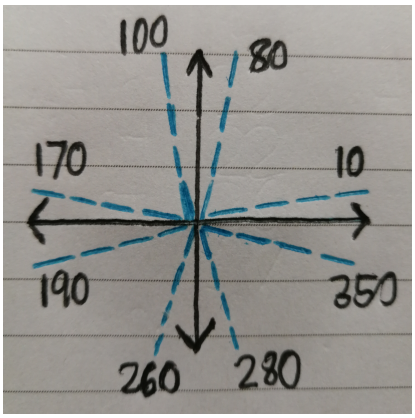

Figure 3.1: Possible Degree Angles Shown in Graph

3.3 Results:

Significant testing is conducted to ensure satisifactory performance by the drone. The referenced table below shows results of the drone visiting sensors on specific days of each year where the month equals the day in the date-specified map. The testing is performed where the drone's starting position is (-3.188396, 55.944425) as indicated in the coursework document requirements.

Date	Moves	Date	Moves	Date	Moves	Date	Moves
01-01-2020	106	07-07-2020	126	01-01-2021	110	07-07-2021	115
02-02-2020	92	08-08-2020	107	02-02-2021	98	08-08-2021	122
03-03-2020	96	09-09-2020	112	03-03-2021	126	09-09-2021	122
04-04-2020	107	10-10-2020	130	04-04-2021	62	10-10-2021	126
05-05-2020	122	11-11-2020	124	05-05-2021	98	11-11-2021	107
06-06-2020	122	12-12-2020	112	06-06-2021	96	12-12-2021	99

The average number of moves made by the drone for the year 2020 is 113 and year 2021 is 106.75 respectively, but 04-04-2021 is clearly an outlier as the sensor points are all within very close distance to each other.

With a regular greedy algorithm, the drone would have normally gotten stuck next to the Appleton Forum and Appleton Tower for the 01-01-2020 and 09-09-2020 map respectively, where the drone would run out of moves. However, my enhanced greedy algorithm solves the problem as described in Section 3.2 if the drone gets stuck. I have explained my examples further using the following figures.

An example where the drone doesn't get stuck is in the date-specified map 2020-03-03 as shown in Figure 3.2. On the other hand, shown in Figure 3.3 the drone was initially stuck but successfully breaks out of the loop by force moving away from the Appleton Forum building in the direction of 10 degree three times before taking readings for the sensors in the map and returning close to its original position.

3.4 Possible Improvements:

While my algorithm does meet the requirements where it does not go outside of the confinement area or passes through any of the No-Fly-Zone areas, the number of moves the drone makes can be reduced. I have discussed the problems with my existing algorithm and possible solutions that could be implemented.

1. **Brute-force checking every direction angle when moving:** It would not be very efficient to check all 35 possible directions to find the minimal distance from its current position to the next sensor point.

   The drone should determine which quadrant would that sensor point should be located in with respect to the drone's position. After finding the quadrant, this method could possibly calculate the angle direction the drone should move towards to by finding the angle between two vectors directly using the formula [4]:

   Let vectors a represent as (x_a, y_a) and b is (x_b, y_b). Let x and y represent the latitude and longitude respectively.

$$\alpha = arccos[x_a * x_b + y_a * y_b)/(\sqrt{((x_a)^2 + (y_a)^2)} * (\sqrt{((x_a)^2 + (y_a)^2)})]$$

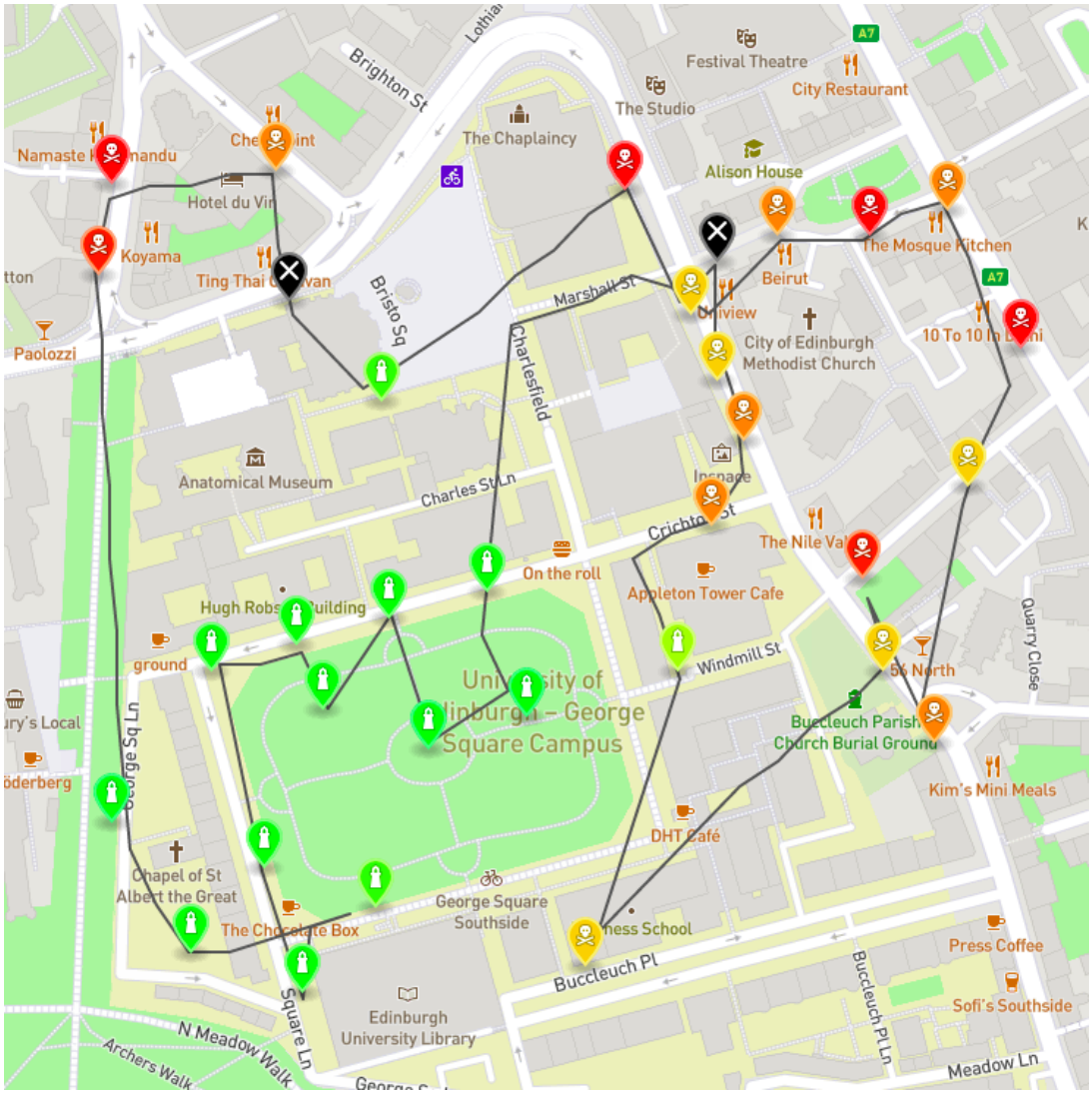

Figure 3.2: 2020-03-03 Map

2. **Making unnecessary moves:** Even though my sub-optimal greedy algorithm has been enhanced to solve the problem if the drone were to get stuck where it would move back and forth, my drone would still need to perform more moves to break out of that loop. In addition, the drone flight paths crossing over one another for some maps clearly indicate there is a more efficient solution.

   The TSP problem could be solved more efficiently if I were to use 2-opt algorithm [2]. This would allow the drone to perform less moves while taking the readings and returning close to its original position. Assuming the map would display 33 sensors, we could create a 33 x 33 distance matrix whose i,j entry is the distance from sensor i to sensor j in degrees. The 2-opt algorithm could be used to swap segments between sensors until the an optimal path is created for the drone to travel while avoiding the No-Fly-Zones. While more complicated, this would ensure that the drone would never get stuck between moving one location to another.

While the average time execution is approximately 1.5 seconds, but the time complexity to continously find the nearest sensor until it has read all sensors is O(n!) as the algorithm is essentially greedy. This could be improved by using the TSP 2-opt solution where the total time complexity including the swaps is $O(n^3)$ [5].

3.5 Conclusion:

While my current algorithm is relatively efficient for the problem, the number of moves the drone could make can be reduced and the efficiency of my code could be further enhanced. Different factors including limited time while maintaining a clean code structure certainly has made implementing this project a tough but fulfilling challenge.

Figure 3.3: 2020-01-01 Map where drone was initially stuck but broke out of loop

3.6 Resources:

1. MapBox Java SDK, https://docs.mapbox.com/android/java/api/libjava-geojson/5.6.0/index.html
   Accessed: 17/10/2020

2. Traveling Salesman Problem Visualization, https://youtu.be/SC5CX8drAtU
   Accessed: 25/10/2020

3. Path2D and Line2D Intersection, https://stackoverflow.com/questions/24645064/how-to-check-if-path2d-intersect-with-line
   Accessed: 19/11/2020

4. Angle Between Two Vectors, https://www.omnicalculator.com/math/angle-between-two-vectors
   Accessed: 03/12/2020

5. A High-Speed 2-Opt TSP Solver for Large Problem Sizes,
   https://on-demand.gputechconf.com/gtc/2014/presentations/S4534-high-speed-2-opt-tsp-solver.pdf
   Accessed: 05/12/2020