ILP Coursework - Report

Matthew Dorling - s1802289

December 4, 2020

1 Software Architecture

1.1 Description

1.1.1 Design Pattern

The design for this piece of software is somewhat based on the Mediator Design Pattern. There is one central class, in this case the `Drone` class that acts as a central point, from which the results of the various operations that together make up the solution to the problem, are centrally managed.

1.1.2 Identification of Operations

The way that I approached the problem was by breaking it down into the different operations that are required to be carried out by the software to achieve a decent solution. The main areas identified are the following:

- Begin the software by interpreting input and running the necessary code
- Interacting with the WebServer (retrieving information)
- Connecting to and retrieving information from the air quality sensors
- Generate a route for the drone to travel around the air quality sensors
- Travelling around the air quality sensors, following the route
- Outputting the necessary information to the output files.

1.1.3 Identification of Classes

These various operations can be designated to specific classes. Giving each class a clear purpose should have the benefit of making the system clean, logical and easy to understand. The following are the classes that carry out the main operations of the program.

- The `App` class
  - Takes the command line arguments and parses them into usable variables.
  - Instantiates the `ServerController`, performs basic checks and instantiates the `Drone` class, passing the `ServerController` over to it.
- The `Drone` class
  - Provides an abstraction of the drone
  - Acts as a Mediator to control the most important classes
  - Instantiate the `Navigator` class to retrieve a route from
  - "Simulate" the movement of the drone around the route, collecting information as it goes and sending it to be formed into output files
- The `Navigator` class
  - Generates a route for the drone to follow to meet the air quality sensors
- the `SensorConnector` class
  - Provides a simulation/abstraction of some physical module on the imaginary drone that would connect wirelessly to the air quality sensor (for example, a Bluetooth module) - in this case we are retrieving the information from a local server.
  - Determines if there is an air quality sensor within range and fetches the data from it
- The `FlightPathFile` class

  - Handles the formatting and writing of the flight path serial file

- The `ReadingsFile` class
  - Collects the readings that the drone retrieves along the route, and the route itself
  - Forms the readings into GeoJson output
  - Writes the output GeoJson file

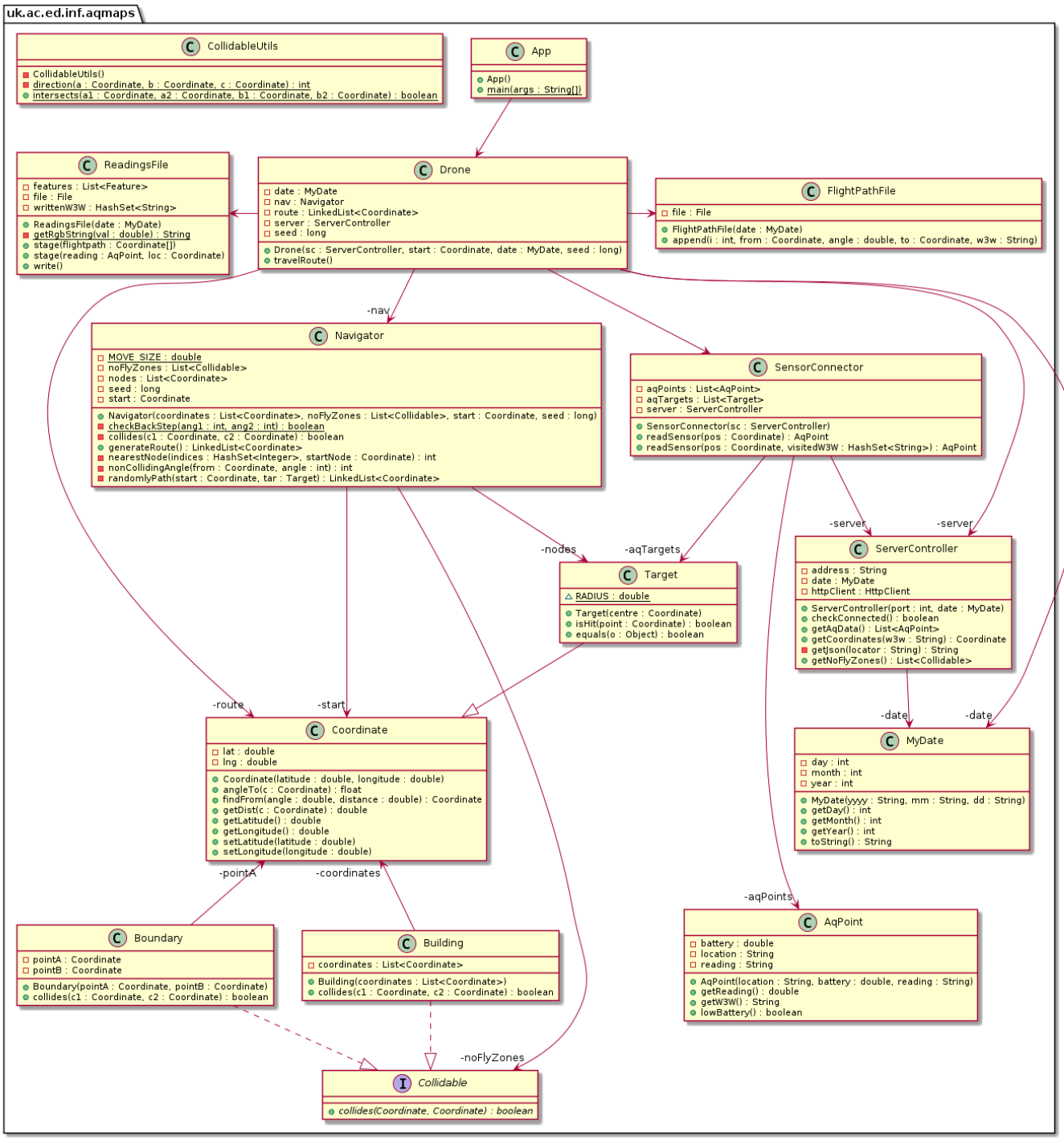
1.1.4 Workflow

The following is a description of the workflow of the software to provide an overview how the main classes interact with each other and the order in which they interact.

1. The `App` class parses the input arguments. It then instantiates a `ServerController`, and the `Drone` class, passing the `ServerController` to it.
2. The `Drone` class instantiates a `Navigator` and calls it to generate a route.
3. The `Navigator` generates a route with the help of private helper methods and smaller methods of the smaller classes such as `Coordinate`, `Target`, the `Collidable` interface and others.
4. The `Drone` then travels along the route. As it travels the route, it collects readings using an instantiation of `SensorConnector`.
5. The `SensorConnector` simulates connecting to any sensor that the drone is in range of after each move and returns the fetched reading information to the `Drone`.
6. The `Drone` uses an instantiation of `FlightPathFile` and an instantiation of `ReadingsFile` to send any the data received from `SensorConnector` and details of each move to be written to the appropriate files.
   - the `FlightPathFile` writes each line to the end of the file for each move after the drone makes a move. It is essentially a serial file.
   - the `ReadingsFile` collects and holds all of the readings data and the flight path because it needs to wait for all the data to be passed to it before it converts the data into a GeoJson string and writes that.
7. The program is finished and ends.

1.2 Class Diagram

The diagram below shows a UML class diagram of the `uk.ac.ed.inf.aqmaps` package[1] [2].



[1] Unfortunately the limitations of A4 sizing means that it was difficult to arrange the class diagram in such a way that it is both laid out clearly and large enough to read. This was the best layout I could come up with. It is easier to read when the pdf is zoomed in slightly.

[2] The diagram was rendered using `https://plantuml.com`

2 Class Documentation

2.1 App

This acts as a starting point to take in parameters and instantiate the `Drone` class.

2.1.1 public static void main(String[] args)

The assumed order of input of `args` is as follows:

- day (DD)
- month (MM)
- year (YYYY)
- starting latitude
- starting longitude
- seed
- port

2.2 Drone

This class is an abstracted representation of the Drone. It mainly interacts with instantiations of the `ServerController` and `Navigator` classes to implement the Drone's function.

2.2.1 public Drone(ServerController sc, Coordinate start, MyDate date, long seed)

Constructor for the `Drone` class. It also instantiates a `Navigator` and loads the route from it.

- `sc` Passes the `ServerController` instantiation to be used by the `Drone` class.
- `start` Passes the `Coordinate` location of the Drone's starting point.
- `date` is the date on which the drone is being used.
- `seed` is the seed used for the randomized pathing algorithm.

2.2.2 public void travelRoute()

This is an abstraction of the Drone's movements along the route that has been generated by the `Navigator` object. It travels the route, checking for sensors after each move using a `SensorConnector`. It uses `FlightPathFile` and `ReadingsFile` instantiations to write the output files as it generates the information.

2.3 Navigator

This class carries out all the algorithms and calculations needed to find a route to navigate to the air quality sensors.

2.3.1 public Navigator(List <Coordinate> coordinates, List <Building> noFlyZones, Coordinate start, long seed)

Constructor for the `Navigator` class.

- `coordinates` is the list of coordinates of the sensors that the drone should visit.
- `noFlyZones` is the list of `Building`s that the that the drone should avoid colliding with.
- `start` is the `Coordinate` location of the Drone's starting point.
- `seed` is the seed that will be used for the randomized pathing.

2.3.2 public LinkedList<Coordinate> generateRoute()

This method uses a nearest-neighbour algorithm to find a route for the drone to travel between all of the sensors. A randomized-pathing algorithm is used to generate moves between the points decided by the nearest-neighbour algorithm, while meeting the constraints such as not hitting buildings and the limited range of motion (i.e. 10 degree compass bearings).
returns the generated route as a linked list of `Coordinate`

2.4 ServerController

This class provides a way of receiving information from the WebServer, which is assumed to be running on `https://localhost/$PORT` where `PORT` is provided as a parameter of the constructor method. It will also load some of the information from the server into necessary objects.

2.4.1 public ServerController(int port, MyDate date)

Constructor for `ServerController` class.

- `port` is the provided port that the WebServer is running on.
- `date` is the provided date that is used to access the correct information from the WebServer.

2.4.2 public String checkConnected()

Checks the server to see if it is possible to connect, whether the given date for the server exists on the server (error 404). Prints a lightweight error message as a string if necessary.
returns true if connection is ok, false if there is a problem.

2.4.3 public List<AqPoint> getAqData()

Finds the air quality data from the `air-quality-data.json` file for the provided date on the server. Parse the json into a List of `AqPoint`.
returns the list of `AqPoint` from the server.

2.4.4 public Coordinate getCoordinates(String w3w)

Find the latitude and longitude of a point at the centre of a What3Words tile, stored on the server.

- `w3w` must be a three words separated by '.'. For example: "slips.mass.baking"

returns a `Coordinate` corresponding to the given w3w parameter.

2.4.5 public List<Building> getNoFlyZones()

Fetch the `no-fly-zones.geojson` file from the WebServer, and parse it into a list of `Building` objects.
returns a list of `Building` objects from the server.

2.5 SensorConnector

This class provides an abstraction of some imaginary module on the drone that would make a connection to a nearby air quality sensor and download the data from it.

2.5.1 public SensorConnector(ServerController sc)

Constructor for the `SensorConnector` class.

- `sc` is a `ServerController` object used by the class to emulate taking readings from air quality sensors.

2.5.2 public AqPoint readSensor(Coordinate pos)

This method checks if there is an air quality sensor near to a given position, and fetches the data from that air quality sensor.

- `pos` is the position to check for nearby sensors from.

returns the data from the sensor as an `AqPoint` object or `null` if there is no nearby sensor.

2.6 Coordinate

Provides a representation of a point on the earth as a pair of degree coordinates and some useful calculations.

2.6.1 public Coordinate(double latitude, double longitude)

Constructor for the `Coordinate` class.

- `latitude` of the coordinate
- `longitude` of the coordinate

2.6.2 public double getDist(Coordinate c)

Calculates the distance from this `Coordinate` to another.

- `c` is the point to which the method will calculate the distance.

returns the distance; the unit is degrees.

2.6.3 public float angleTo(Coordinate c)

Calculates the angle from this `Coordinate` to another.

- `c` is the point to which the method will calculate the angle.

returns the distance; the unit is degrees and the value will be between -180 and +180. [3]

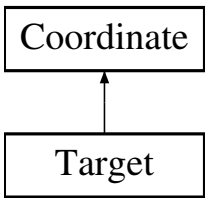
2.6.4 public Coordinate findFrom(double angle, double distance)

Calculates a point from this `Coordinate`, given the angle and the distance to it.

- `angle` is the angle to the next point.
- `distance` is the distance to the next point.

returns the location of the new point.

2.7 Target



This class extends `Coordinate`, adding a radius. These are used to represent the sensor points while generating a route.

2.7.1 public Target(Coordinate centre)

Constructor for the class. Essentially transforms a `Coordinate` into a `Target`.

[3] This is converted to 0 to 360 elsewhere in the project for output

2.7.2 public boolean isHit(Coordinate point)

Checks whether a given point is within the radius of the target. The radius is the specified constant of 0.0002 (in degrees)
returns true if point is within radius, false if not.

2.7.3 public boolean equals(Object o)

Overrides the default `equals` method to provide functionality for checking if two Targets are the same.

2.8 Collidable

An **interface** to define an object or physical space that the drone should not collide with.

2.9 Building

This class implements `Collidable` and provides an abstraction of a building - a polygon within the map with which the drone cannot collide.

2.9.1 public Building(List<Coordinate> coordinates)

Constructor for the `Building` class.

- `coordinates` is a list of points defining the perimeter of the building.

2.9.2 public boolean collides(Coordinate c1, Coordinate c2)

Overrides. Determines whether a move from one point to another will collide with the building.

- `c1` and `c2` are the start and ending points of a move.

returns true if the move collides, false if not.

2.10 Boundary

This class implements Collidable and provides an abstraction of a boundary - one side of the drone confinement area.

2.10.1 public Boundary(Coordinate pointA, Coordinate pointB)

Constructor for the `Boundary` class.

- `pointA` and `pointB` are the endpoints of the boundary line.

2.10.2 public boolean collides(Coordinate c1, Coordinate c2)

Overrides. Determines whether a move from one point to another will collide with the boundary.

- `c1` and `c2` are the start and ending points of a move.

returns true if the move collides, false if not.

2.11 CollidableUtils

This class provides functionality to determine if a Collidable object has been collided with.

2.11.1 public static boolean intersects(Coordinate a1, Coordinate a2, Coordinate b1, Coordinate b2)

Method to determine if the line between two points intersects the line between another two points.

- `a1` is first point on first line.
- `a2` is second point on first line.
- `b1` is first point on second line.
- `b2` is second point on second line.

returns true if intersects, false if not.

2.12 MyDate

Simple class to represent a date: year, month and day. Much simpler to use for this project than any pre-written Date implementations.

2.12.1 public MyDate(String yyyy, String mm, String dd)

Constructor for the `MyDate` class.

- `yyyy` is a string of 4 digits representing the year
- `mm` is a string of 2 digits representing the month (i.e. 03)
- `dd` is a string of 2 digits representing the day (i.e. 03)

2.12.2 public String toString()

Creates a string from the day, month, and year.
returns a string in the form `"dd-mm-yyyy"`

2.13 FlightPathFile

This class handles the preparation and writing of the flightpath output text file.

2.13.1 public FlightPathFile(MyDate date)

Constructor for the `FlightPathFile` class. Checks if the file already exists. If so, delete it in order to replace it. Create the file.

- `date` is used to write the file with the correct filename.

2.13.2 public void append(int i, Coordinate from, double angle, Coordinate to, String w3w)

This method takes all of the required information of a move for one line of the output file, converts it into an appropriate string for the line, then appends it to the file.
This file is essentially a serial file so we append each line one at a time with this method.

- `i` is the integer indicating which move it is.
- `from` is the starting position of the move.
- `angle` is the angle of the move.
- `to` is the ending position of the move.
- `w3w` is the What3Words position of any detected sensor. If there is no detected sensor, this should be `"null"`

2.14 ReadingsFile

This class handles the preparation and writing of the readings output geojson file. The geojson file requires all parts of the file to be ready before writing, so each part of the file is staged as it is made available to the instantiation of this class. The file can be written once all parts are staged.

2.14.1 public ReadingsFile(MyDate date)

Constructor for the `ReadingsFile` class. Checks if the file already exists. If so, delete it in order to replace it. Create the file.

- `date` is used to write the file with the correct filename.

2.14.2 public void stage(Coordinate[] flightpath)

This method creates a `LineString Feature` representing the drone's flightpath, and adds it to a list of `Features` to be written later.

- `flightpath` is an array of the positions that the drone takes on its flight, to be written as a GeoJson LineString.

2.14.3 public void stage(AqPoint r, Coordinate loc)

This method creates a `Marker Feature` representing the reading collected from an air quality sensor, and adds it to a list of `Features` to be written later.

- `reading` is the `AqPoint` reading collected from an air quality sensor that will be staged for writing.
- `loc` is the location of the sensor from which the reading was taken.

2.14.4 public void write()

This method writes all of the previously staged Features to the GeoJson file.

2.15 AqPoint

This class provides a simple way of storing the information of an air quality sensor as an object.

2.15.1 public AqPoint(String location, double battery, String reading)

Constructor for the `AqPoint` class.

- `location` is the What3Words location
- `battery` is the value indicating the battery level percentage
- `reading` is the value indicating the air quality reading

2.15.2 public boolean lowBattery()

returns true if the battery is below 10%, false otherwise

3 Algorithm

3.1 Overview

For my implementation of this task, I split the drone control algorithm into two parts:

- Generating a route
- Following the route

Generating the route presented a need for more complex algorithms. I split up this part into two main problems:

3.1.1 Deciding in which order to visit the sensors

This is essentially a variation of the Travelling Salesman problem. For this part, I decided to use a nearest-neighbour style algorithm. It is one of the most commonly used algorithms for these kinds of problems. These algorithms are fairly straightforward to understand and also has a reasonable computational complexity for the task.

3.1.2 Determining a path between each sensor

This aspect of the algorithm was perhaps the most challenging to find a solution for, due to the constraints imposed upon the drone. The following are the main constraints:

- Only able to travel on compass bearings that are multiples of 10 degrees
- Not colliding with buildings
- Not leaving the square drone confinement area
- Only being able to take one reading after any one move

For this part, I based my algorithm upon the principles of randomized algorithms[4]. It perhaps has the most similarities with Las Vegas algorithms[5], a type of randomized algorithm; it always gives a correct solution or it fails (rarely). In the case of a fail, the routing algorithm will instead path to the next nearest sensor.
With the various constraints on how the drone can move, it would be difficult to mathematically generate the best path to the next sensor, and it would be too computationally complex to use another algorithm such as a brute-force algorithm.

The random algorithm calculates the angle from the current position to the target sensor, adds a (controlled) random offset to vary the direction slightly, and repeats that until it has either succeeded or failed finding the target sensor. Repeat until a suitable path has been found.

This is an heuristic algorithm. It will not find the best possible path, but will find a good and useable path.

[4] `https://brilliant.org/wiki/randomized-algorithms-overview/`
[5] `https://en.wikipedia.org/wiki/Las_Vegas_algorithm`

3.2 Simplified Workflow of Algorithms

- Starting at the start position, find the nearest sensor position (from server)
- Try to path from the current position using the randomized pathing algorithm to that sensor
    1. Get the angle as a multiple of 10 degrees that points closest to the target
    2. Add a randomized bias (multiple of 10 degrees) within a certain range
    3. Adjust the angle by adding +/- 10 degrees until not hitting any building or edge of confinement area
    4. Calculate the next position of using that angle from the current position
    5. Repeat 1 and 2 until hitting only the desired target.
    6. If there is a problem with that route, try starting again from the current position
- Repeat these until all sensors have been visited.
- Use randomized pathing to path back to start.
- Drone travels that entire route, up to 150 moves:
    1. For each move in the route:
    2. Move the drone to the next position
    3. Take a reading from any nearby sensor
    4. Send reading data to be written to file

3.3 Pseudocode of Algorithms

nodes is an attribute, a list containing the points of all the sensors that need to be visited

3.3.1 generateRoute

- **unvisited** ← new Hash set of integers
  // stores indices of unvisited nodes
- **path** ← new linked list of Coordinates
  // stores the positions of each point on the path
- **pathToAppend** // define list to store section of path that will be appended
- **unvisited** ← add all indices of nodes (ie. $\{0..n-1\}$ where n is the length of **nodes**)
- **path**.add the drone starting position
- do while there are unvisited indices
  - **targetIndex** ← *getIndexNearestNode*(unvisited, current position)
  - **targetNode** ← get **node** at **targetIndex**
  - **failedVisits** ← new stack of integers
    // this will store any indices where the random pathing fails
  - do while **pathToAppend** is empty // it is empty when *randomlyPath* has failed
    - **pathToAppend** ← *randomlyPath(currentPos, targetNode)*)
    - if randomlyPath failed and **unvisited** contains other indices
      - **failedVisits**.add the failed **targetIndex**
      - **unvisited**.remove the failed **targetIndex** ← *nearestNode(unvisited, currentPos)*
        // get index of next nearest node
    - else if randomlyPath failed but there are no other unvisited indices // very rare case
      - clear **unvisited** and **failedVisits**
      - exit this loop
  - pop any indices from **failedVisits**, add back into **unvisited**
- **pathToAppend** ← *randomlyPath(currentPos, startPos)*
  // try to path back to starting position
- **path**.add **pathToAppend**

3.3.2 getIndexNearestNode(unvisited nodes indices, dronePos)

- **shortestDist** ← max possible value
- **nearestIndex** ← 0
- for each unvisited index // from parameter
  - check if node at the index is closer than the closest
  - if it is, update **shortestDist** and **nearestIndex**
- return **nearestIndex**

3.3.3 randomlyPath(startPos, target)

- **rand** ← random number generator with seed
- **attemptCounter** ← 0
- do while no successful route yet found
  - **attemptCounter2** ← 0
  - **path** ← new linked list
  - increment **attemptCounter** by 1
  - if **attemptCounter** > 500, return **path**
    // prevent infinite loops (unlikely to be used, but not impossible)
  - do while route has not hit the target or become trapped/stuck
    - if **attemptCounter2** > 150, return **path**
      // prevent infinite loops (unlikely to be used, but not impossible)
    - **ang** ← the angle from last position in **path** to the target
    - if the last position is closer than 2 x the move size, angle range is +/- 90 degrees, else +/- 30 degrees
    - **randAng** ← random multiple of 10 within that range
    - **ang** ← **ang** + **randAng**
    - adjust **ang** for collisions with noflyzones by adding +/- 10 degree increments until it does not collide
    - **path**.add calculate drone's next position using **ang** from the current position and move size
    - if **ang** and the previous angle are in opposite direction: route has become stuck
    - if target has been hit, exit this loop
  - if the target has been hit and no other targets have been hit and the route has not become stuck, exit the loop.
  - else, try to find a path again.
- return **path**

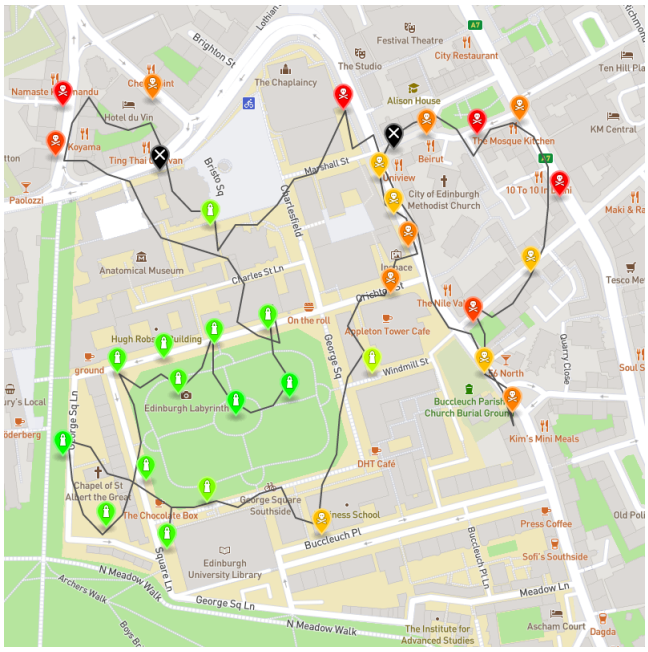
3.3.4 travelRoute

// this is the algorithm for the drone to follow the generated route, collecting readings.

- **route** holds the route previously generated by generateRoute()
- **moves** ← length of **route**
- if **moves** > 150, **moves** = 149
- for **i** = 0 to **moves** - 1
  - **previousPos** = **route**[i]
  - **currentPos** = **route**[i+1]
  - read sensor at **currentPos**
  - if no sensor is detected, use What3Words = "null"
  - send any reading data and position to file objects to be written

3.4 Sample Map Images

The following are images for 03/03/2020 and 08/08/2020



3.5 Review

Overall, the algorithm works successfully. I used a bash script to run it over all the days over the two years that are included on the server. In total, that takes around 14 to 17 minutes on my computer which would indicate a running time of around 1 to 1.5 seconds per day. Furthermore, the way I have implemented the nearest neighbour means in the future this algorithm could be upgraded to a more optimal Travelling Salesman Problem algorithm, even though this one works well.