Informatics Large Practical Report

Patryk Marciniak
s1828233

Contents

Software Architecture Description

General Structure: My *aqmaps* project consists of 7 separate Java classes: *App, Location, Sensor, ServerParser, DrawMap, Drone, DroneMoves.* I identified these classes as being the correct ones, because they all serve a separate purpose and are all very important to make sure the program fulfills the task it was designed to do, i.e. “program an autonomous drone which will collect readings from air quality sensors distributed around an urban geographical area” - source ilp-coursework specification.

The **App** class acts as the entry point of the application through the *main* method which parses command line arguments, and passes the appropriate information to the respective classes that make the application function correctly. It is also responsible for writing the results to the two output files.

The **Location** class is used to create a new object that represents a coordinate location consisting of two double values, a latitude and longitude. It also contains methods that use Location objects, these methods range from calculating new locations from existing ones, to finding the distance between two locations.

The **Sensor** class is used to create a Sensor object representing an air quality map sensor, consisting of 3 variables, a String *reading,* double *battery* and String *Location.*

The **ServerParser** class is used to establish a connection to a local web server and read data from the web server and parse it into the appropriate data structures. Specifically it is used to parse the list of no-fly-zones, the list of the 33 sensors to visit and the coordinate and what3words string locations.

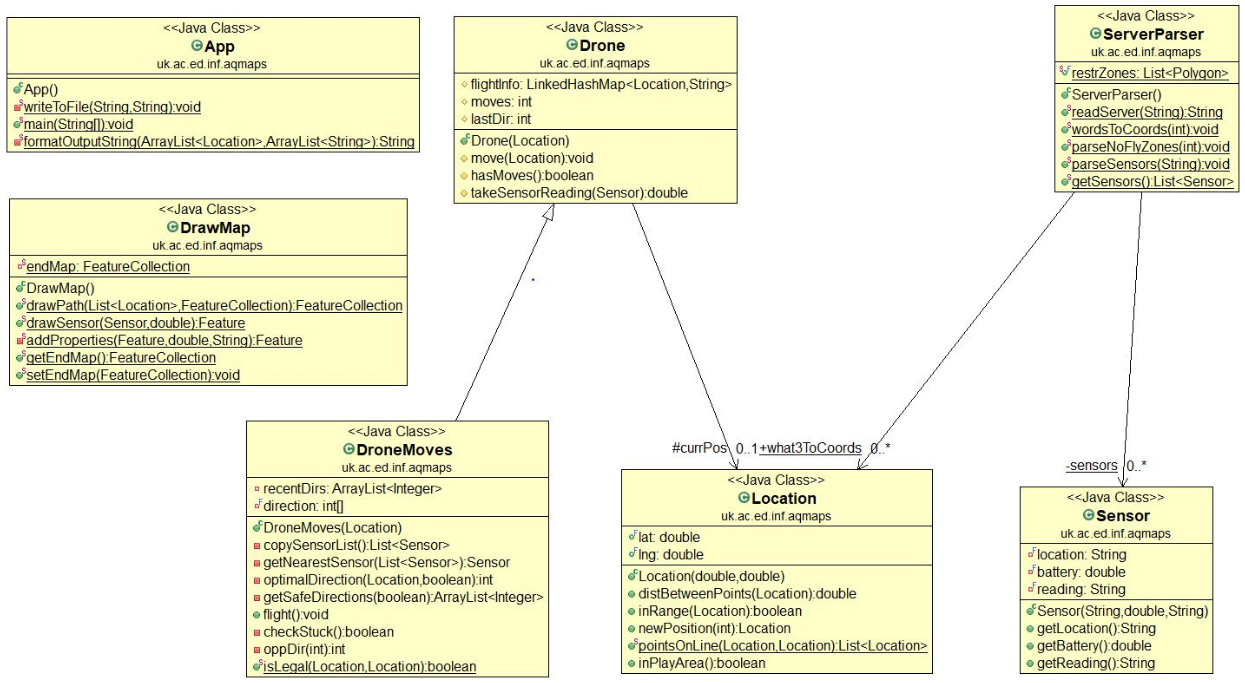
The **DrawMap** class is used to add various features to the feature collection that will be used to generate the final GeoJson map. Specifically, the class is used to add the path taken by the drone and the sensors visited to the *endMap*, and to create the appropriate symbol for the sensor, depending on the reading the drone received from it.

The **Drone** class is used to create a drone object, which contains a Location variable, representing the drones current position and is used to keep track of the number of moves the drone has taken and the last direction it moved in.

The **DroneMoves** class is a subclass of **Drone** used to extend it’s functionality to be able to control how the drone actually moves around a map.

UML Diagram

The UML diagram below demonstrates the hierarchical relationship between the classes mentioned above.



<<Java Class>>
App
uk.ac.ed.inf.aqmaps
App()
writeToFile(String,String):void
main(String[]):void
formatOutputString(ArrayList<Location>,ArrayList<String>):String
<<Java Class>>
Drone
uk.ac.ed.inf.aqmaps
flightInfo: LinkedHashMap<Location,String>
moves: int
lastDir: int
Drone(Location)
move(Location):void
hasMoves():boolean
takeSensorReading(Sensor):double
<<Java Class>>
ServerParser
uk.ac.ed.inf.aqmaps
restrZones: List<Polygon>
ServerParser()
readServer(String):String
wordsToCoords(int):void
parseNoFlyZones(int):void
parseSensors(String):void
getSensors():List<Sensor>
<<Java Class>>
DrawMap
uk.ac.ed.inf.aqmaps
endMap: FeatureCollection
DrawMap()
drawPath(List<Location>,FeatureCollection):FeatureCollection
drawSensor(Sensor,double):Feature
addProperties(Feature,double,String):Feature
getEndMap():FeatureCollection
setEndMap(FeatureCollection):void
<<Java Class>>
DroneMoves
uk.ac.ed.inf.aqmaps
recentDirs: ArrayList<Integer>
direction: int[]
DroneMoves(Location)
copySensorList():List<Sensor>
getNearestSensor(List<Sensor>):Sensor
optimalDirection(Location,boolean):int
getSafeDirections(boolean):ArrayList<Integer>
flight():void
checkStuck():boolean
oppDir(int):int
isLegal(Location,Location):boolean
#currPos 0..1
+what3ToCoords 0..*
<<Java Class>>
Location
uk.ac.ed.inf.aqmaps
lat: double
lng: double
Location(double,double)
distBetweenPoints(Location):double
inRange(Location):boolean
newPosition(int):Location
pointsOnLine(Location,Location):List<Location>
inPlayArea():boolean
-sensors 0..*
<<Java Class>>
Sensor
uk.ac.ed.inf.aqmaps
location: String
battery: double
reading: String
Sensor(String,double,String)
getLocation():String
getBattery():double
getReading():String

This UML diagram was generated using Eclipse's ObjectAid Plugin

As can be seen from the UML diagram above. The classes have relatively loose coupling(measure of how closely connected two classes are- lower is better) between them and can be considered in a linear relationship, since bar **App** and **DrawMap** each class connects to one other class. Also since each class serves its own purpose and contains methods which serve that purpose, the code also has high cohesion(degree to which the elements inside a class belong together-higher is better) within classes.

The above is based on knowledge learned in the Year 2 Course- Software Engineering. See reference 1.1.

Class Documentation

App:

Methods:

- String writeToFile(String *fileName,* String *str*): Writes the String *str* to a file called *fileName*.
- String formatOutputString(ArrayList<Location>*path*, ArrayList<String>*details*): Uses the values in *path* and *details* to format the final output string containing all the information about positions of the drone at each move, directions taken and sensors visited. It then returns this final string.
- *String main (args[])*: Parses the command line arguments and makes sure they are valid. It then generates the appropriate URI for the map using the input values for day, month, year, port. Using this address it calls the **ServerParser** method that parses the sensors that need to be visited for that day. It then calls two further methods from the class **ServerParser** which parse the locations for the sensors parsed earlier and parse the restricted polygons that the drone cannot fly over. Then the main method creates an instance of **DroneMoves** class and calls it's flight method which generates the drones path. Then by calling the drawPath() method, the method adds the generated flight path to the FeatureCollection *endMap*. Now it calls the method formatOutputString which formats the string that will be written to the text file. Finally the method writes the FeatureCollection string to a geoJson file, and the output string to a text file.

***Location*:**

Class Variables:

- final double *lat*: The latitudinal position of the Location object
- final double *lat*: The longitudinal position of the Location object

The above variables are immutable because a location doesn't change once it's created.
The constructor in the class uses the *lat, lng* variables to initialize a Location object.

Methods:

- Double distBetweenPoints(Location *newPos*): An instance method which returns the distance between the Location object it's called on and the *newPos* as a value of type double. The distance between the two Locations is found using the Pythagorean distance formula.

- Boolean inRange(Location *newPos*): Checks if the Location the method is called on is within 0.0002 degrees of *newPos.* If it is, method returns true, if not returns false.
- Location newPosition(int *degrees*): Using trigonometry, calculates a new position 0.0003 degrees away at an angle of *degrees* from the Location it's called on. It then returns this new position as a new Location object.
  Note: This method follows the convention that 0° means go East, 90° means go North, 180° means go West, and 270° means go South.
- List<Location>pointsOnLine(Location *start*, Location *end*): Given a *start* and *end* location, returns a list of 1000(1002 including *start* and *end*) points at equal intervals on the straight line going from *start* to *end*.
  Note: This method was adapted from a method found on a website, see reference 1.3.
- Boolean inPlayArea(): Checks the Location the method is called on is within the bounds of the 'play' area. If it is, method returns true, else returns false.

***<u>Sensor</u>*:**

Class Variables:

- final String *location* - The what3words string location of the sensor
- final double *battery* - The battery level of the sensor
- final String *reading* - The air quality reading of the sensor

The above variables are all immutable because once the values of the sensor are set, they do not change throughout the execution of the program.
The constructor then uses the above 3 variables to create a Sensor object.

Methods:

Getters for the 3 class variables: getLocation(), getBattery(), getReading().
Note: Since Sensor variables are immutable and will get their data from the web server, setters are not required.

***<u>ServerParser</u>*:**

Class Variables:

- static final List<Polygon> *restrZones*: The list of Polygons that the drone cannot fly over
- static final HashMap<String, Location> *what3ToCoords*: Holds the what3word string and its corresponding Location
- static List<Sensor> *sensors*: The list of sensors that need to be visited

The above variables are static, because the data they hold is the same for all objects and instances.

Methods:

- String readServer(String *uri*): Establishes a connection to a local web server found at the specified *uri.* Then reads data on the server at that address and converts the data into a string and returns it.
- Void wordsToCoords(int *port*): Loops through all of the stations to be visited and for each use the *port* and the what3words String location of the Sensor to read the corresponding coordinates for that sensor of the web server. It then creates a new Location object from those coordinates and adds the what3words String and the Location to *words3ToCoords.*

  Note: The part of this method that deals with creating a Location object from the web server data was taken from stackOverflow post. See reference 1.2
- Void parseNoFlyZones(int *port*): Uses port to generate a URI which directs the readServer method to where the data on the no fly zones is held. The method then takes the String returned by readServer, and converts it into FeatureCollection. It then deserialises the Feature Collection into a list of Features and then into separate Polygons which are then added to *restrZones.*
- Void parseSensors(String *uri*): takes in a *uri* which will be generated in the App class from the command line arguments, which will direct the readServer method to where the list of sensors that are to be visited on that day are held. It then parses String returned by read server into a list of Sensor objects.

***<u>DrawMap</u>*:**

Class Variables:

- static FeatureCollection *endMap*: The final map which will contain all the visited sensors and the path the drone took to visit the sensors.

Methods:

- FeatureCollection drawPath(List<Location>*path,* FeatureCollection *map*): Creates a linestring from all the locations in *path*, by converting each location into a point, then the list of points into a linestring. This linestring is then converted into a feature and is added to the FeatureCollection *map*.
- Feature drawSensor(Location *sensorPos,* double *reading*): Using the position of the sensor *sensorPos,* creates a feature at that location and calls the colorSensor method to add the properties to it. It then returns this as a feature.
- Feature addProperties(Feature *ftr,* double *reading*, String *location*): Takes a feature *ftr* representing a sensor on a map, and adds the appropriate color and symbol to it depending on it’s *reading.* It also adds a location property to the feature, contained in the string *location* parameter.
- Getter and setter for *endMap*, used to get and set the FeatureCollection *endMap.*

***<u>Drone</u>*:**
Class Variables:

- LinkedHashMap<Location,String> *flightInfo*: LinkedHashMap in order to maintain order of elements, which contains information regarding the drone's flight path, with the Location part representing the locations the drone has been to and the String holding information about the move number and direction taken to get to the corresponding location.
- Location *currPos*: Location representing the drone's current position
- int *moves*: The number of moves the drone has made
- int *lastDir*: The last direction the drone has moved in.

Note: the above variables are all 'protected' so that the **DroneMoves** class has full access to them

The drone's current position(*currPos*) is used in the constructor of the class to initialize a **Drone** object.

Methods:

- void move(Location *newPos*): This method is responsible for moving the drone. It takes in a location, and updates the drones *currPos* to *newPos* and increases the drone move count by 1
- boolean hasMoves(): this uses the drones *moves variable* to check that the drone has not exceeded its maximum move allocation of 150, if it has it the method returns false if not it returns true.
- double takeSensorReading(Sensor *sensor*): This method takes in a *sensor*, then checks it's battery level, if it is less than 10%, it returns -1 to signify the sensor's reading is voided due to having low battery. If the battery is 10% or more, the method uses **Sensor's** getReading method, and returns the sensor's reading.

***<u>DroneMoves</u>*:**

Class Variables:

- ArrayList<Integer> recentDirs: The list of directions, the drone has moved in since it was last stuck.
- final int[] direction: The immutable list of all possible directions in degrees, that the drone can move in, i.e. only multiples of 10, between 0 and 350 degrees.

Methods:

- List<Sensor> copySensorList(): This method creates a deep copy of the list of Sensors, found in the Sensor class.
- Sensor getNearestSensor(List<Sensor> *leftToVisit*): This method finds the nearest sensor to it's current position from the list of sensors that the drone has not yet visited *leftToVisit*. It does this using a basic finding minimum algorithm which will go through all the sensors in *leftToVisit* and calculate the distance from the drone's *currPos* to each of the sensors and will return the Sensor that is closest.
- ArrayList<Integer> getSafeDirections(boolean *stuck*): Goes through all the possible directions the drone can move in from *direction*. For each direction calculates a new position using **Locations** newPosition(), if *stuck* is false, the new position must satisfy isLegal(), inPlayArea() and must not be directly opposite to the *lastDir* taken by the drone, then that direction is added to a list of safe directions. If *stuck* is true, an additional parameter must be satisfied by the new position, which is that the direction cannot be part of *recentDirs*.
  The method then returns an ArrayList *safeDirs* which holds all the safe directions the drone can move in from it's current position.
  Note: it finds the opposite direction using the *oppDir* method.
- int optimalDirection(Location *targetPos*, boolean *stuck*): First this method calls getSafeDirections(), and passes in it's *stuck* parameter, to get a list of all safe directions on the given move. Then out of those safe directions for the drone to take, this method finds the optimal direction, which if taken will take the drone's currPos, closest to the *targetPos.* It does this once again with a modified finding minimum algorithm, which will calculate a new position based on each direction and use the *distBetweenPoints* method of the **Location** class to find the distance between the new position and *targetPos.*
- boolean checkStuck(): This method uses the class variable *recentDirs* to check if the drone is stuck. First it removes all but the last 4 (If it has that many) elements of *recentDirs*, so that only the most recent directions taken are considered. It then loops through *recentDirs* and for each direction it uses the *oppDir method* to find it's opposite direction and checks if at any point two consecutive directions in *recentDirs* are directly opposite each other +/- 10 degrees - implying the drone is zigzagging aimlessly. Every time the method detects this, it increments the *repeat* counter by 1. If this counter exceeds 2, the method declares the drone as stuck and returns true.
- int oppDir(int dir): This method simply calculates and returns the direction directly opposite to the input direction *dir.*
- boolean isLegal(Location *startPos,* Location *endPos*): This method returns true if the path going from *startPos* to *endPos* is legal, i.e. meaning it doesn't cross any

of the no fly zones and is within the play area. To check if the move is legal, the Location class *pointsOnLine* method is called to generate a list of 1000 Locations(points) on the path going from *startPos* to *endPos.* The method will now loop through all the points and create each as a MapBox point, so that it can check if the point is inside any of the no fly zones represented as polygons stored in **ServerParser's** *restrZones* variable. To check if a point is inside a polygon, Mapboxes Turf library is used, in order to access it's TurfJoins classes inside method-which takes in a point and a polygon and returns true if the point is inside the polygon. If for all points on the line this returns false, the path is declared legal and the method returns true, if TurfJoins.inside detects any point inside the polygon, the method returns false.

Note: A Mapbox Turf dependency was added to the project so the Turf library could be used in the above method.

- void flight(): This is the most important method of the **DroneMoves** class as it uses all the other methods in the class to create a flight path for the drone and populates the **Drone** classes *flightInfo* with the details of the generated flight path.
The aim of this method is to generate a path for the drone that tries to visit 33 sensors and take their readings to add them to a FeatureCollection which will represent the final map and then will try take the drone back to where it started initially within 150 moves. It will first call the copySensorList() method to generate a copy of the list of sensors to visit. Then while the drone has moves left and the number of sensors to visit is > 0 it will:
    1. Find the nearest station to the sensor using getNearestSensor().
    2. Use the optimalDirection() method to find the best direction to get to that nearest sensor.
    3. Uses checkStuck() to see if the drone is stuck. If it is, it will calculate the next best direction found by optimalDirection() and clears *recentDIrs*. If the drone is not stuck, the method continues to step 4.
    4. Generate a string *output*, which will contain the move number, the direction taken on that move and either the word *null* if no sensor was visited on that move, or the what3words string location of the sensor that was visited.
    5. Checks if the drone is within range of the sensor using inRange(), if this returns true:
        - The method call's takeSensorReading() to get the reading of the sensor
        - It will now call drawSensor() from **DrawMap** to create a feature at the given sensors location and add the appropriate properties to it

        depending on the reading. Which will either be the reading of the sensor if it was charged or -1 if it had low battery. It then adds the feature created by drawSensor *sensor* to the method variable *feats*-representing a list of features.
    - Removes the visited sensor for the list *leftToVisit* so that the drone doesn't waste moves trying to get to a sensor that has already been visited.

    If inRange() is false the method proceeds to the next step.

6. Add the drones *currPoss* and string *output* of the current moves details to the Drone's *flightInfo* variable
7. Use the Locations classes *newPosition*() with the optimal direction found to calculate the drone's new position from it's *currPos*
8. Move the drone to the *newPos*
9. Repeat above steps while hasMoves() is true and not all sensors have been visited.

If all sensors have been visited and the drone still hasMoves(), it will repeat step 2,3,4,6 except it will try to find the best direction to get back to the starting position and check if the drone is inRange() of the starting position. If it is, the boolean variable *finished* is set to true, ending the drones movement. If not it repeats steps 7,8.
It will keep repeating the above process until the drone has either moved to within the starting position, or until hasMoves() is false.

Once the drone's path generation is complete and *flightInfo* is populated. The method carries out a final check, in case the drone was not able to visit all 33 stations within it's allocated moves,i.e. If the drones moves are equal to 150 but the number of stations left to visit is > 0. If this returns true, for each of the remaining sensors to visit, the drawSensor method will be called, with the reading parameter at -2, meaning the sensor has been unvisited and should be coloured grey. Each of the features returned by drawSensor is then added to *feats.*
Finally the method calls setEndMap() and passes in the FeatureCollection formed from *feats.*

Drone Algorithm:

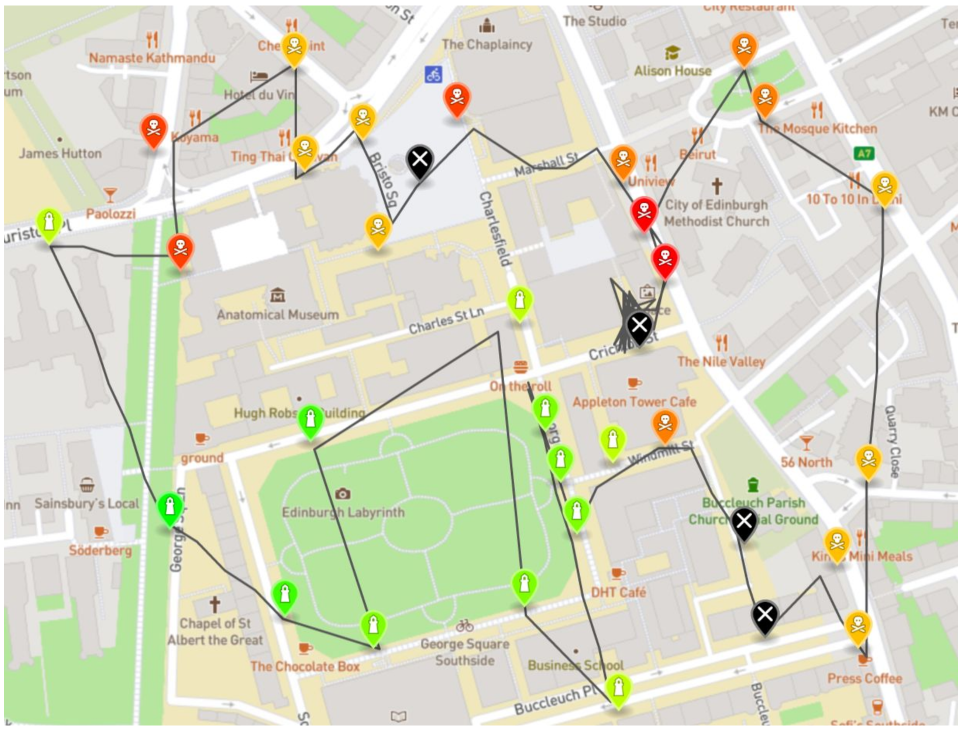
The basic idea of the drone algorithm is to follow a greedy approach(see reference 1.4), meaning the drone will always try to go to the nearest sensor to it’s current position, based on Pythagorean distance. Although multiple modifications have been made in order to get around the no fly zones.

The drone starts by locating the nearest sensor to it’s starting position it then finds all the legal directions available to the drone from it’s current position and finds the best direction to get there and then moves towards it, even if it is initially in range of the sensor. Once the drone is within the sensor (< 0.0002 degrees), it takes the reading of the sensor and processes it to update that sensor's status on the map. It then removes this sensor from the list of sensors to visit, so that the drone doesn’t end up wasting moves on trying to visit a sensor that has already had it’s data read. It then proceeds to find the next nearest unvisited sensor and moves towards it. The drone algorithm will keep repeating the above steps until the list of sensors to visit is empty or until it has run out of it’s allocation of 150 moves.
Now that all sensors have been read, if the drone still has some moves remaining, it will try to return to within 0.0002 degrees of it’s original starting position. Using the same strategy as above, except ignoring all sensors and only trying to get back to start.
Once the drone has visited all the sensors and returned to it’s starting position or once it has reached 150 moves, the drone algorithm stops the drone and completes it’s movement on the map.

The modifications to the above explained greedy approach are:
In order to ensure the drone doesn’t go over any no fly zones zones, the drone algorithm first pre-calculates the next position of the drone if it were to take the optimal direction to the target. It then checks 1000 points(1002 including start,end points) between the line from the drones current position to the estimated new position. This is to ensure that at no point in the expected move does the drone cross over a no fly zone. If none of the points are detected inside any no fly zone, then the move is legal and the drone carries it out.

To make sure the drone doesn’t get stuck in between buildings or anywhere else, the algorithm maintains a list of recent directions, which contains the last 4 directions the drone has moved in since the drone was last stuck. In order to check if the drone is stuck, before making a move the algorithm will go through the list of recent directions, and each time it finds two consecutive directions that are directly opposite +/- 10 degrees(meaning the drone is zigzagging aimlessly) it increments a counter monitoring repeated moves. If this counter reaches 3, the algorithm declares the drone as stuck. If the drone is declared stuck the algorithm will make the drone 'unstuck' by picking the next best safe optimal direction that isn't the one that resulted in the drone being stuck. The algorithm assumes that this new direction will make the drone unstuck so it clears the list of recent directions taken.

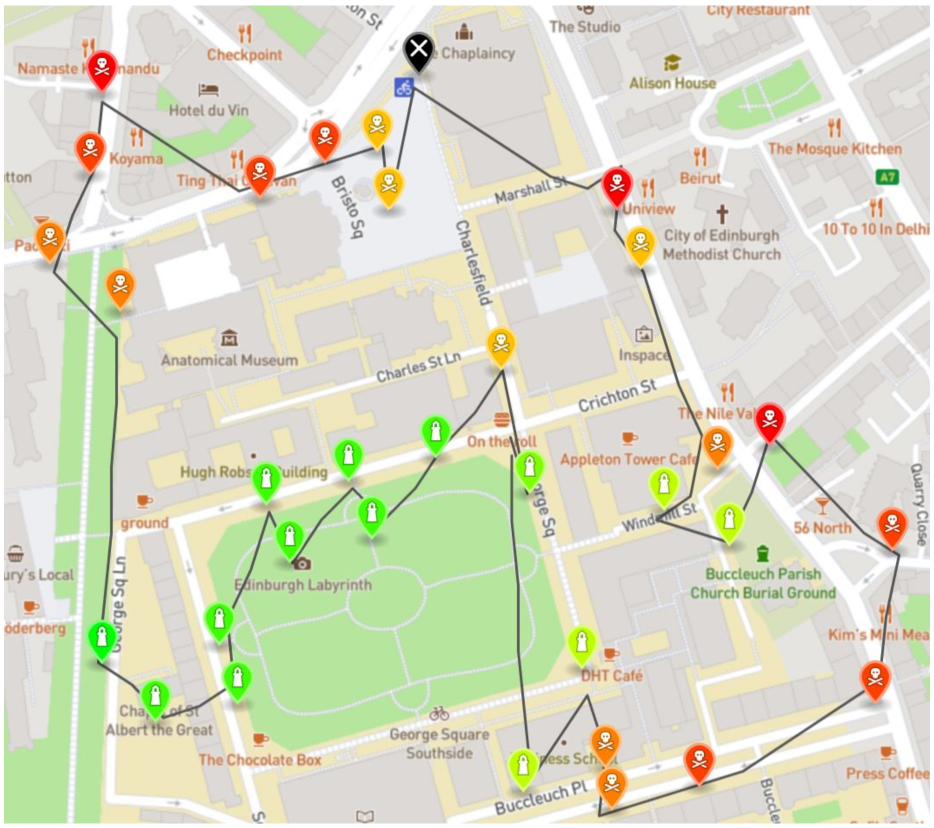
Sample Maps



Sample result map for day 09/09/2020, with starting position 55.9444 -3.1878, completed in 116 moves

This map demonstrates a weakness of my algorithm. Due to the fact that the drone bases it's best direction to move in only on the information it has at any one step and as a result is not 'aware' of the rest of the map, the optimal direction can end up taking the drone into a gap in the no fly zones, which it gets temporarily stuck in(as demonstrated above) so it ends up wasting a lot of moves, going backwards and forwards until it can get out of this 'gap'.

A possible improvement would be to make the algorithm consider a few moves in advance so it doesn't lead to the case above, although this would come at the expense of time, as more things would need to be computed and the algorithm would be slower.



Sample map for 21/02/2020 with starting position 55.9444 -3.1878, completes in 79 moves

On the other hand this map shows a strength of this greedy approach algorithm. When the sensors are all laid out in a 'loop' like above, the optimal direction picked will result in a short path, because the drone won't have to worry about getting stuck on or between buildings. Another benefit of my greedy based algorithm is that it doesn't need to do much computation on each step, meaning it is relatively fast.

References:

1.1. UML diagram and basic knowledge about software architecture and desirable traits come from things learnt in Inf2C- Software Engineering

1.2 Stack Overflow post on deserializing Json
https://stackoverflow.com/questions/64537086/parsing-specific-elements-of-a-json-string-in-java
Credit to Michał Ziober who explains how to deserialize specific parts of a Json string in his comment.
Date accessed:22/10/2020

1.3 Finding points on line adapted from:
https://dzone.com/articles/how-find-point-coordinates
Date accessed: 07/11/2020

1.4 Greedy algorithm:
https://brilliant.org/wiki/greedy-algorithm/#:~:text=A%20greedy%20algorithm%20is%20a,to%20solve%20the%20entire%20problem
Date accessed:28/10/2020