Informatics Large Practical

Coursework 2

Report

- **S1892592**

Table of Contents

1. Overview:

The task for the Informatics Large Practical is to program an autonomous drone which will collect readings from air quality sensors distributed around an urban geographical area as part of a (fictitious) research project to analyse urban air quality. There are 4 no fly zone buildings - Appleton Tower, Informatics Forum, David Hume Tower and the Main Library over which the drone is not allowed to fly or cross.

In all, there are 99 sensors distributed around the University of Edinburgh's Central Area and only a list of 33 sensors is produced which enumerates the sensors which need to be read for a day. While collecting the readings from the sensors, the drone is not allowed to go outside the Drone Confinement Zone or cross any of the No Fly Zone buildings.

The drone flight path should be a closed loop, which means after it has collected the readings from all the 33 sensors for that respective day it is supposed to come back to the starting position.

2. Software architecture description:

In order to create a coherent structure there's a class for each important aspect in the problem domain. Separation of implementation is being used to split up the program into different classes. Each of the class implements a specific function. This helps in testing specific parts of the code instead of running the entire program just to check a few lines of code. It also increases the *cohesion* and decreases the *coupling* of the code making it easier for the other person to understand the code and modify (or maintain) the implementation according to his/her likeness.

The program can be divided into 3 parts:

1. Main control
2. Features
3. Tests

These three are further divided into 6 java classes including the App class.

The **Main Control** consists:

1. App

The **Features** consists:

1. Drone
2. Position
3. Sensor
4. WhatThreeWord

The **Tests** consists:

1. AppTest

All the classes are correlated to each other which prevents duplication of the code and helps in easier implementation of the desired task.

The **App** class serves as the main control of the entire framework and is the only entry point to the program through the *main()* method . It first checks the validation of the arguments entered in the command line, for example check if the given starting position of the drone is inside the Drone Confinement Zone or not. Then it sets up a connection with the HTTP-Client web server to parse the GeoJson and Json files like *no-fly-zones.geojson*, *air-quality-data.json* and first/second/third/details.json. After parsing the files it instantiates the Drone class and calls its methods to generate a permutation of sensors as they are visited by the drone. At last it generates the text output file for the reading taken by the drone and the GeoJson file for the flight path followed by the drone to visit all the sensors and then return back to the starting point.

The **Drone** class is responsible for the movement of the drone and collection of the reading from the sensors while avoiding the No Fly Zones. It also makes sure that the drone comes back to the starting point after all the 33 sensors are read for that particular date and all of its moves are inside the Drone Confinement Zone.

The **Position** class represents the position of the drone as the assignment is position based hence we have this class. It has 2 attributes - lng (longitude) and lat (latitude). It also consists of some helper methods like *distance()* - find the distance between the 2 positions, *findClosestSensor()* - find the sensor closest to the current position of the drone, *containPoint()*, *insideDroneConfinementZone()* and *findDirection()*.

The **Sensor** class represents a sensor, it is needed to encapsulate all the relevant fields of the sensor like - location, battery, reading, position, etc. Therefore it also has some getters and setters for the private fields of the class. It consists of a helper method *inRange()* - check if the current position of the drone is in range of that sensor.

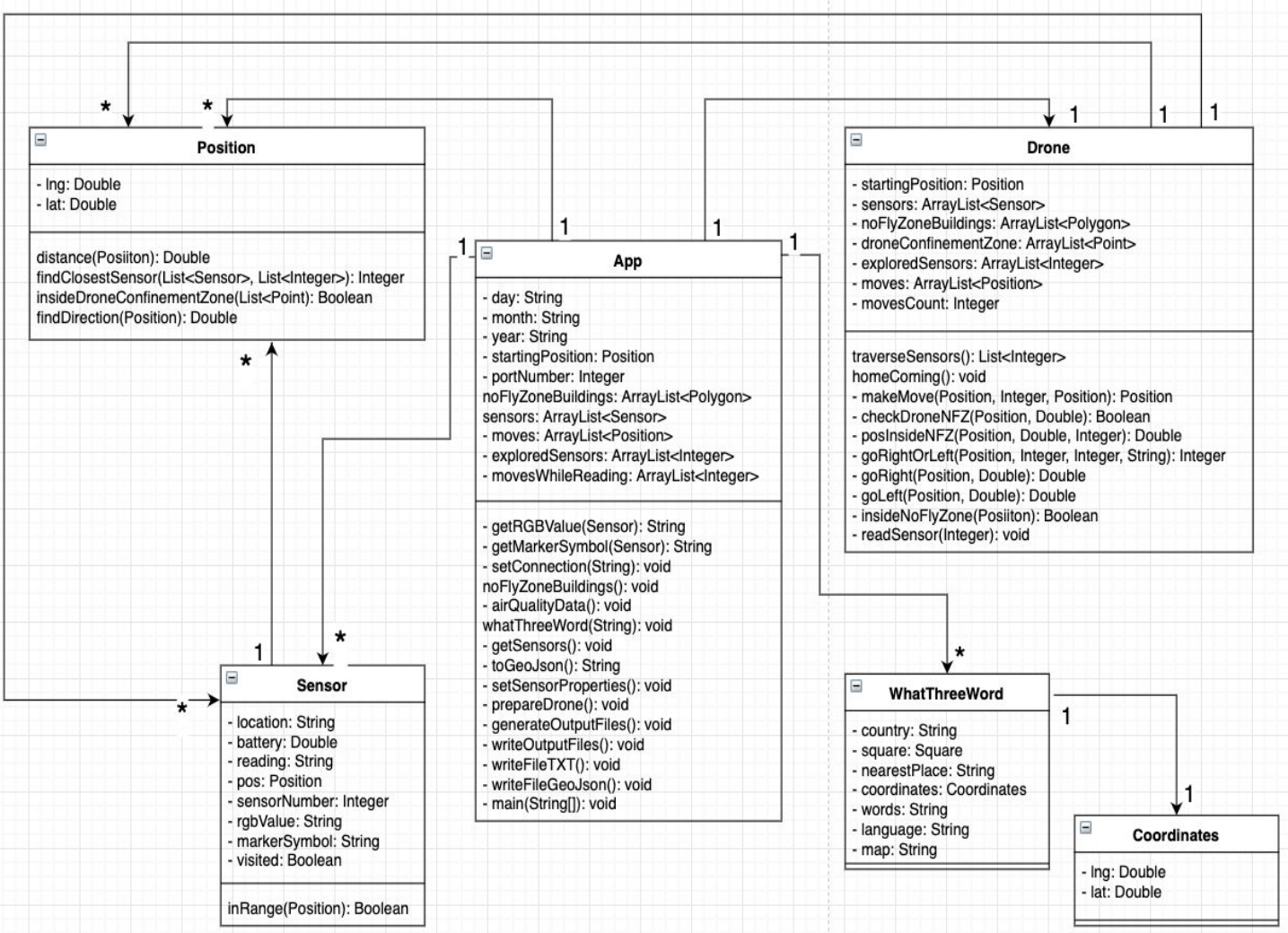
The **WhatThreeWord** class is to parse the Json file for location of the sensor. It consists of attributes like coordinates - coordinates of the sensor, country, square, words, etc. Along with the getter and setters for the attributes of the class.

The **AppTest** class is to test the output files generated by the **App** class for the given date.
It consists of two method -
testTxtOutputFiles() which checks:
direction <= 350 degrees, no negative direction, the length of one move is 0.0003 degrees, whether the drone is in range of the sensor when it's reading one (distance < 0.0002), number of lines in the file <= 150 and the number of sensors read == 33.
Then finally delete the generated txt file.

testGeoJsonOutputFiles() which checks:
number of features in the file == 34, number of points == 33 and
the number of line strings == 1.
Then finally delete the generated GeoJson file.

UML Class Diagram:

In this part of the report I'll show some hierarchical relationships between the classes and also give a brief glance of the methods and variables present in each class along with their scope of access.



3. ***Class Documentation:***

- App.java

final static private Point ***Forrest_Hill*** **-** The *final* variable to initialise the coordinates of *Forrest Hill* (edge of the the drone confinement zone)

final static private Point ***KFC*** **-** The *final* variable to initialise the coordinates of *KFC* (edge of the the drone confinement zone)

final static private Point ***Buccleuch_St_bus_stop*** - The *final* variable to initialise the coordinates of *Buccleuch_St_bus_stop* (edge of the the drone confinement zone)

final static private Point ***Meadows_Top*** - The *final* variable to initialise the coordinates of *Meadows_Top* (edge of the the drone confinement zone)

private List<Point> droneConfinementZone - List of edges of the drone confinement zone.

private String day, **private** String month, **private** String year - Date for which the drone has to read the sensors.

private Position startingPosition - Starting position of the drone.

private int portNumber - Port number of the web server.

List<Polygon> noFlyZoneBuildings - GeoJSON polygons of No Fly Zone Buildings.

List<Sensor> sensors - List of sensors to be read.

private List<Position> moves - List of moves made by the drone.

private List<Integer> exploredSensors - List of sensors already visited.

private List<Integer> moveWhileReading - Indexes of the move when drone reads a sensor

Methods:

public static void main(String[] args) **throws** IOException, InterruptedException

The *main()* method first checks if the arguments entered by the command line are in correct format. Then it instantiates the App class to call its respective methods to make the drone fly around all the sensors and take readings. After the drone has visited all the sensors the main method calls another method to generate the text and geoJson output files.

public App(String day, String month, String yr, Position startPos, **int** portNumber)

The constructor instantiates the global variables of the App class.

private Object setConnection(String urlString) **throws** IOException, InterruptedException

The *setConnection()* method sets the connection with the HTTP web server and gets the response from the file at address *urlString*.

void noFlyZoneBuildings() **throws** IOException, InterruptedException

The *noFlyZoneBuildings()* method parses the geoJson file for the No Fly Zone buildings on the web server.

private void airQualityData() **throws** IOException, InterruptedException

The *airQualityData()* method parses the Json file for the air quality data and gets the list of details about all the 33 sensors to be read that particular day.

WhatThreeWord whatThreeWord(String loc) **throws** IOException, InterruptedException

The *whatThreeWord()* method calls the *setConnection()* method to set up the HTTP client connection and then parses the Json file for the location of the sensor (three words) and returns an instance of that class.

private void getSensors() **throws** IOException, InterruptedException

The *getSensors()* method adds other properties to the instances of sensors in the list generated by the *airQualityData()* method such as its position in (lng, lat) by calling the method *whatThreeWord()*.

private void prepareDrone()

The *prepareDrone()* method instantiates the **Drone** class and calls its methods to traverse over all the sensors and then come back to the starting position.

private String toGeoJson()

The *toGeoJson()* method creates and returns the featureCollection of all the features to be written in the output file like the sensors and the line string showing how the drone visited them.

private void setSensorProperties()

The *setSensorProperties()* sets the properties for all the sensors like the color and the markerSymbol by calling methods *getRGBValue()* and *getMarkerSysmbol()* respectively based on the sensor's reading.

private String getRGBValue(Sensor sensor)

The *getRGBValue()* method returns the rgb value for that sensor according to its reading.

private String getMarkerSymbol(Sensor sensor)

The *getMarkerSysmbol()* method returns the marker symbol for that sensor according to its reading.

private void generateOutputFiles()

The *generateOutputFiles()* method generates the output files by creating them and then calling the method - *writeOutputFiles()* to write inside them.

private void writeOutputFiles(String fileNameTXT, String fileNameGeoJson)

The *writeOutputFiles()* method opens the files and call methods - *writeFileTXT()* and *writeFileGeoJson()* to write the moves made by the drone and the featureCollection of all the features respectively.

- Drone.java

private Position startingPosition - Starting position of the drone.

List<Sensor> sensors - List of sensors to be read.

List<Polygon> noFlyZoneBuildings - GeoJSON polygons of No Fly Zone Buildings.

private List<Point> droneConfinementZone - List of edges of the drone confinement zone.

private List<Integer> exploredSensors - List of sensors already visited.

private List<Position> moves - List of moves made by the drone.

private int movesCount
Count of moves made by the drone.

Methods:

public Drone(Position startingPosition, List<Sensor> sensors, List<Polygon> noFlyZoneBuildings, List<Point> droneConfinementZone)

The constructor initialises the global variables of the class.
List<Integer> traverseSensors()

The *traverseSensors()* method makes the drone traverse over all the sensors avoiding the No Fly Zone in 150 moves.

void homeComing()

The *homeComing()* method makes the drone return back to the starting point of the drone after all the sensors are read, avoiding the No Fly Zones.

private Position makeMove(Position currentPos, **int** closestSensorNum, Position target)

The *makeMove()* method receives the current position of the drone, the number of the closest sensor and target (if the drone is heading back to the starting position) to generate the next position of the drone and then calls method *checkDroneCrossNFZ()* to check if its outside the No Fly Zone and inside the Drone Confinement Zone.

private boolean checkDroneCrossNFZ(Position currentPos, **double** direction)

The *checkDroneCrossNFZ()* method checks if the drone's next position in the direction received is inside the No Fly Zone or not, by breaking a move of 0.0003 degrees into 1000 smaller moves (or points) and checking if any of them lies in the No Fly Zone and returns true if it lies inside and false otherwise.

private double posInsideNFZ(Position currentPos, **double** direction, **int** closestSensorNum)

The *posInsideNFZ() method* is called by the *makeMove()* method if the position of the drone lies inside the No Fly Zone. This method further calls the method *goRightOrLeft()* twice to get the number of moves that drone would take if it goes from the left side of the No Fly Zone building and by the right side of the No Fly Zone building. Considering the less number of moves it decides whether to go left or right and further calls methods *goLeft()* and *goRight()* respectively.

private int goRightOrLeft(Position currentPos, **int** closestSensorNum, **int** movesCountTemp, String sign)

The *goRightOrLeft()* is a recursive method which receives the current position of the drone, the number of moves already made by the drone and the sign (whether to increase (+) or decrease (-) the direction by 10 degreed). It finds the next position of the drone in the new direction by either increasing or decreasing the direction based on the sign received then calls itself by passing the new position of the drone, the same sensor number and increment moves count and the same sign received.

private double goRight(Position currentPos, **double** direction)

The *goRight()* method is called by the method - *posInsideNFZ()* when the drone is supposed to go from the right side of the No Fly Zone building. It keeps on incrementing the direction by 10 degrees till it finds a new direction in which the drone doesn't cross the No Fly Zone and returns the new direction.

private double goLeft(Position currentPos, **double** direction)

The *goLeft()* method is called by the method - *posInsideNFZ()* when the drone is supposed to go from the left side of the No Fly Zone building. It keeps on decreasing the direction by 10 degrees till it finds a new direction in which the drone doesn't cross any of the No Fly Zone and returns the new direction.

private void readSensor(**int** closestSensorNum)

The method *readSensor()* is called by the method *traverseSensors()* when the drone is in range of that sensor. This method adds the sensor number to the explored list.

private Boolean insideNoFlyZone(Position pos)

The method *insideNoFlyZone()* receives the position of the drone and checks if that position is inside any of the no fly zone buildings or not.

- Position.java

private double lng - Longitude
private double lat - Latitude

Methods:

public Position(**double** longitude, **double** latitude)

The constructor initialises the global variables (i.e lng and lat) of the **Position** class.

double distance(Position pos)

The *distance()* method calculates the distance between two points by using the Pythagorean distance.

int findClosestSensor(List<Sensor> sensors, List<Integer> exploredList)

The *findClosestSensor()* method finds the closest sensor to the current position of the drone which hasn't been already explored by using the method *distance()*.

boolean insideDroneConfinementZone(List <Point> droneConfinementZone)

The *insideDroneConfinementZone()* method checks if the position used to call the method is inside the Drone Confinement Zone or not.

double findDirection(Position pos)

The *findDirection()* method computes the direction between the position it's called with and the position received in the parameters.

- Sensor.java

private String location - Location of the sensor (in what 3 word format)

private double battery - Battery of the sensor

private String reading - Reading of air quality

private Position pos - Position of the sensor (in longitude and latitude format)

private int sensorNumber - Number of the sensor

private String rgbValue - rgb color for the sensor based on its reading

private String markerSymbol - marker symbol for the sensor based on its reading

private boolean visited - true if the sensor is visited by the drone, false otherwise.

Methods:

boolean inRange(Position position)

The *inRange()* method checks if the current position of the drone (received in parameters) is in range of the sensor (<= 0.0002 degrees) or not.

4. Drone control algorithm:

Approach:

The main algorithm used in the program is the **Greedy Algorithm**, in which the drone finds the closest sensor to its current position and keeps on moving towards it until it comes in range of that sensor. Once it's in range of the sensor, it reads the sensor and continues finding another closest sensor. This approach reduces the number of moves by a significant amount as it finds the closest sensor instead of following a predefined permutation.

Algorithm:

Once it finds the closest sensor, the drone finds the direction in which it is supposed to head to get to that sensor if the direction is not in multiples of 10 the program rounds it off to the closest multiple of 10 and accordingly calculates the next position of the drone (0.0003 degrees away). This continues till all the 33 sensors are read and the number of moves made by the drone are less than 150.

Finding next position:

To find the next position of the drone the program first computes the next position in the direction of the sensor, if that position crosses any of the no fly zones the *posInsideNFZ()*

method is called which further calls the recursive method - *goRightOrLeft()* twice to find the number of moves the drone would take to fly around the no fly zone building from both its sides (left and right) to get in range of the sensor, then selects the direction with the less number of moves and returns a new direction in which the drone should move following to which the next position is computed in the new direction. This makes the drone even more efficient as it finds the shortest path around the no fly zone building.

Making a move:

After the final next position (outside the No Fly Zone) for the drone is computed the drone moves to that position, the current position is changed to the new position, the moves made by the drone is incremented and that move is added to the *moves* list which keeps track of the moves made by the drone.

Dealing the No Fly Zones:

Incase the new position in the direction of the closest sensor lies inside or crosses any of the No Fly Zones then the program continues to find the new direction by increasing and decreasing the original direction by 10 degrees until the next position is outside the no fly zone then it further calculates the number of moves it would take to get to the sensor from the right side and the left side of the no fly zone building and selects the direction with less number of moves. After computing the next position a check is made if that position is inside the drone confinement zone and not crossing any no fly zone.

Checking if the drone is crossing No Fly Zone:

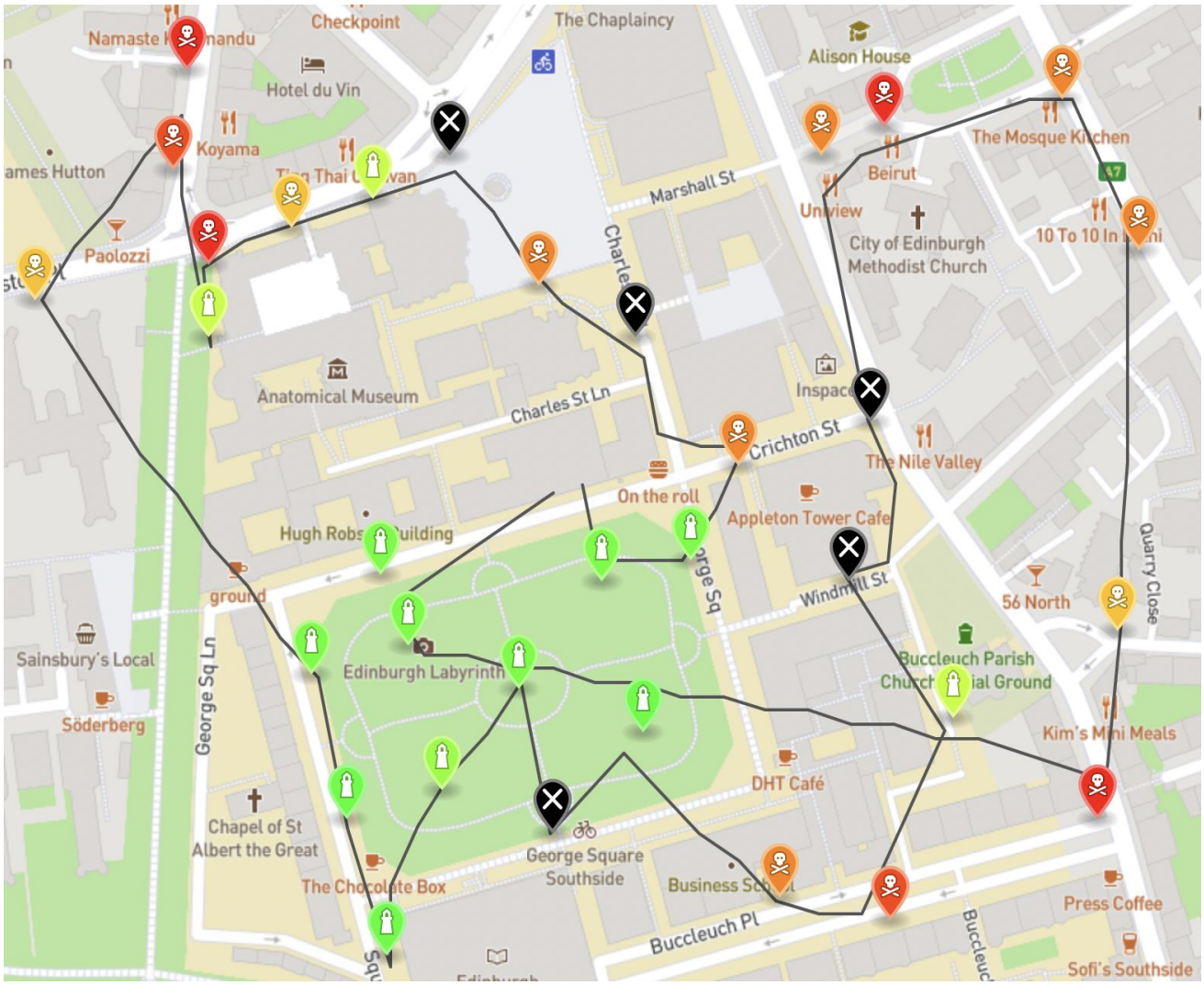
Everytime before the next position of the drone is computed *checkDroneCrossNFZ()* method is called which checks if the next position of the drone in the given direction lies inside or crosses any of the No Fly Zone buildings (Appleton Tower, Informatics Forum, David Hume Tower and Main Library). This method breaks one move of 0.0003 degrees into 1000 smaller points of 0.0000003 degrees and checks if any of the points on the line to the next position lies inside the No Fly Zone or not. It returns a true if it crosses the no fly zone and a false otherwise.

Reading the sensor:

Once the drone comes in range of a sensor it takes the readings from it and the sensor number is added to the explored list of sensors in order to make sure it's not visited again.

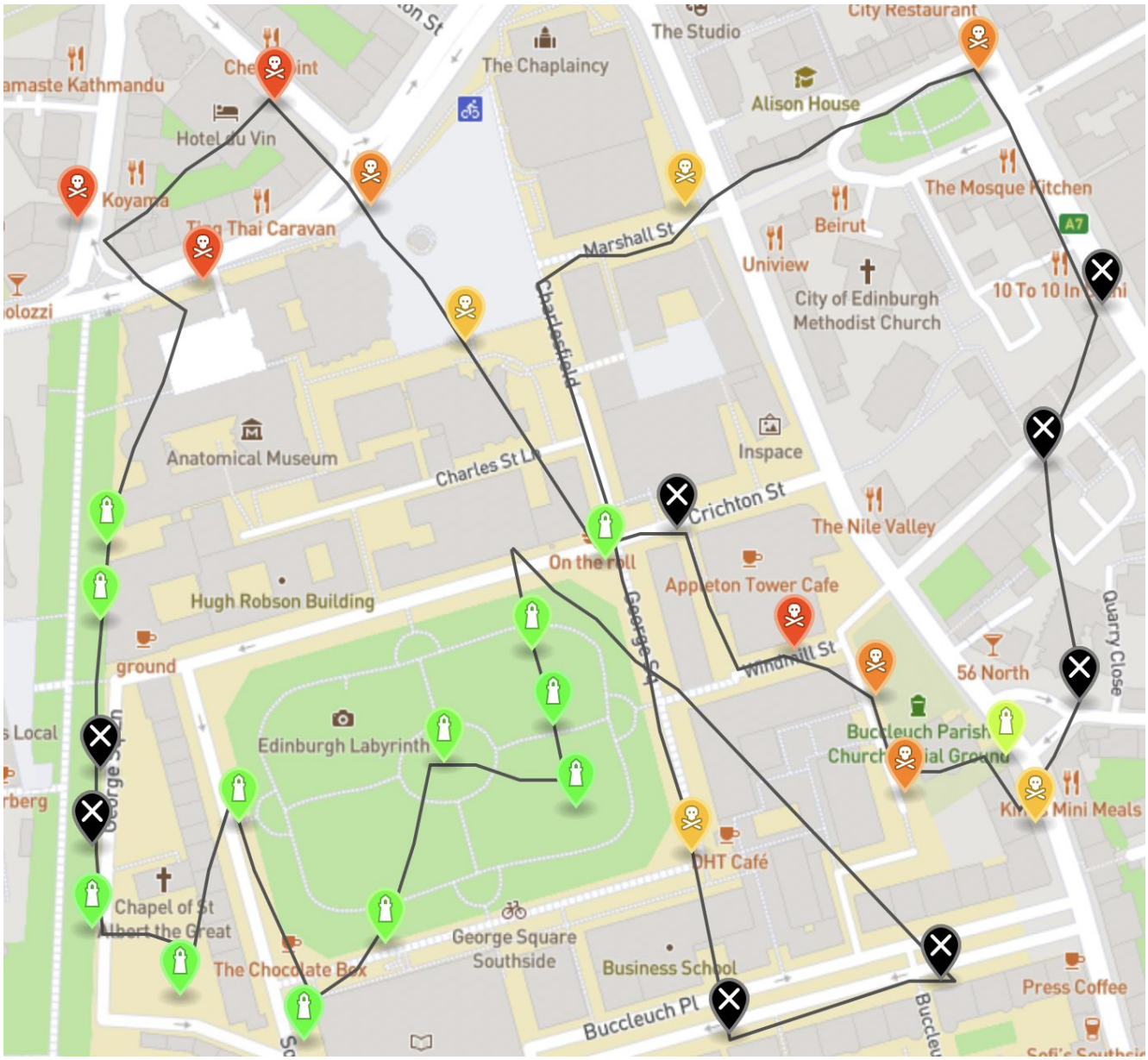
Coming back to starting location:

After the drone has read all the 33 sensors, *homeComing()* method is called which brings back the drone close to the starting position avoiding the no fly zones within 150 moves.



- Flightpath for date - *13/06/2020* with starting point *(-3.188396, 55.944425)*

For the map of **13th of June 2020**, the drone algorithm takes **102 moves** to read all the **33 sensors** and then return back to the starting location. The permutation in which the sensors are explored are : 5, 29, 22, 17, 23, 13, 27, 25, 30, 20, 26, 21, 15, 14, 16, 18, 0, 9, 6, 24, 11, 7, 12, 1, 32, 4, 8, 10, 19, 3, 2, 28, 31 (with 0 as the first sensor and 32 as the last in the list of sensors).



- Flightpath for date - *13/07/2020* with starting point *(-3.188396, 55.944425)*

For the map of **13th July 2020**, the drone algorithm takes **98 moves** to read all the **33 sensors** and then return back to the starting location. The permutation in which the sensors are explored are : 20, 27, 5, 4, 12, 2, 11, 6, 7, 22, 17, 18, 30, 13, 29, 8, 0, 21, 24, 25, 23, 14, 31, 10, 28, 26, 3, 19, 15, 16, 32, 1, 9 (with 0 as the first sensor and 32 as the last in the list of sensors).