ILP report

Coursework 2

Huacheng Song

s1826390

Overview:

This is the final report for ILP. This report contains three sections, they are:

- Section 1: Software architecture description.
- Section 2: Class documentation.
- Section 3: Drone control algorithm.

1. Software architecture description

1.1 UML Class Diagram

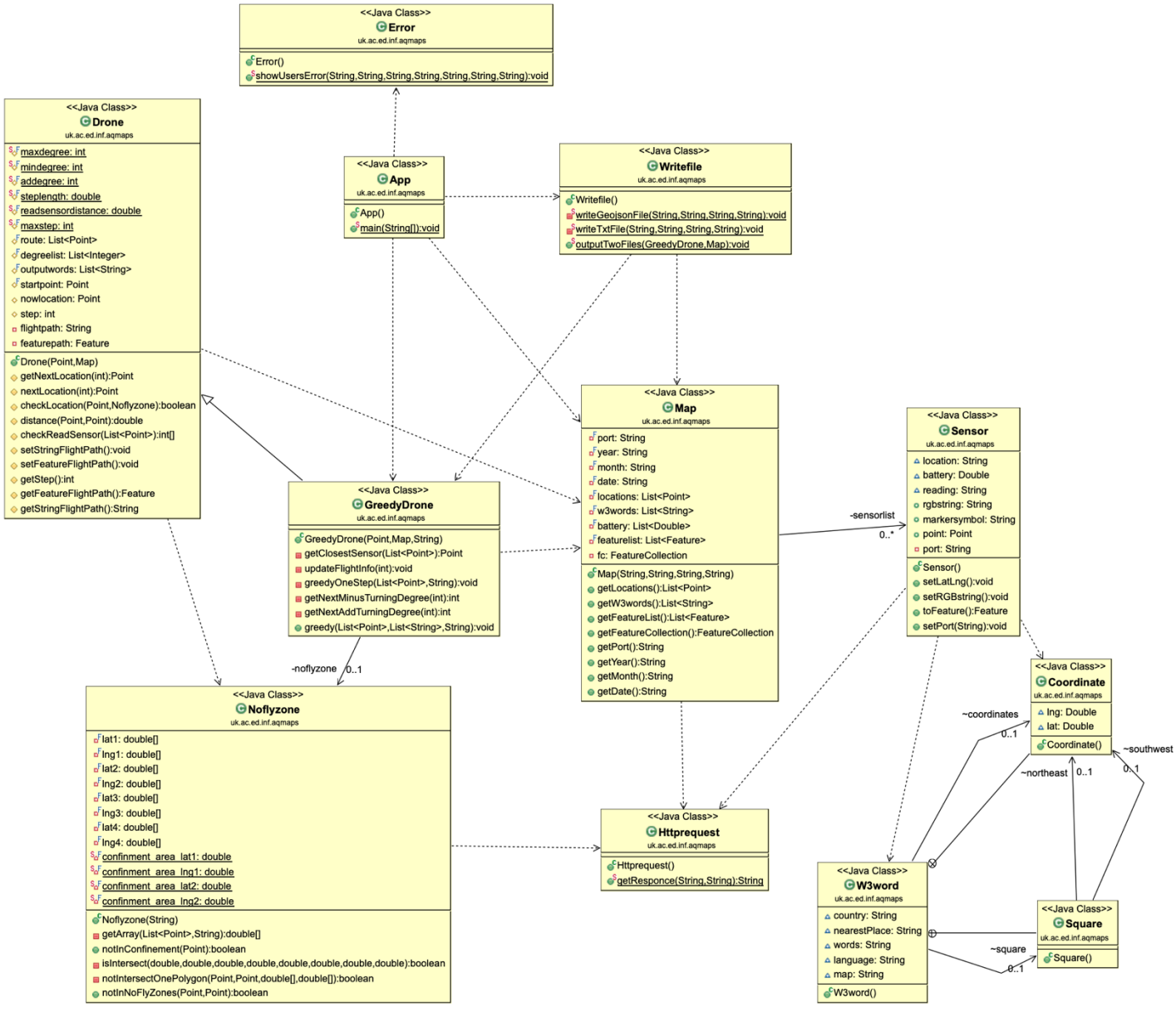

Figure 1: UML Class Diagram

1.2 Classes description

The UML class diagram above indicates the class structure of the application which consist by 12 classes: App, Drone, GreedyDrone, Map, Sensor, W3word, Square, Coordinate, Noflyzone, Httprequest, Writefile and Error.

The below classes convert the entire application into pieces which make the whole program tidy and structural.

App. App class only contains the main function which will hand over works to other classes to complete. App class only call other classes methods, which keeps the class simple and clean. App class contains the entire work, when someone reads the code in App class, they can find that the code is logical and easy to understand as they are all named methods, which shows the running sequence of the entire program.
We identify this class because it is the entrance of the entire application and the program runs from here.

Drone. Drone class shows how the drone work and it contains all the basic operation of the drone, such as calculating the distance and flying to the next point. Drone class is the super class of GreedyDrone class.
We identify this class because it does not contain any algorithms related to drones, hence when we need to write some algorithms for drones, there is no need to change anything in this class and we only need to extend this class. This makes the whole task extremely tidy and efficient.

GreedyDrone. GreedyDrone class extends by the Drone class. It extracts data information from Drone class and generate output corresponding with the Greedy algorithm for the program. GreedyDrone is in the first two important class (the other is Map class) in the whole program to a large extent as it has the most interactions with other classes.
We identify this class because next time when we want to change some part of the algorithm, we only need to change in this class. Thus, it is separated from Drone class. Using class hierarchy makes the algorithm part structural and independent of the entire program.

Map. A date parses argument input. Map class is fundamental and important to the whole program. Map class contains all the information of air quality data on a speciated date, such as location, battery and the corresponding w3word of each sensor. And the drone will extract data from the map in order to find a route.
We identify this class to store the map information of a speciated date and could easily extract any sensor information and this allows the drone to operate with only this class and a start point.

Sensor. Sensor class is necessary since the Json need to be parsed in order to get each sensor information.
We need Sensor class to store each sensor information.

W3Word. Although Sensor class contains all the information of sensor, the location of sensor is given in a W3Word format which is a String, but we need the actual coordinates for future calculation.
Hence W3Words class is identified for parsing W3Word details and W3Word class is necessary.

Square & Coordinate. Square class and Coordinate class are the subclasses in W3Word class. Coordinate is nested by additional brackets in Square and Square is nested by additional brackets in W3Words.
We need to define these classes in order to parsing W3Word from the webserver. Square class and Coordinate class are necessary.

Noflyzone. There are four no-fly-zones in the campus and there is a confinement area which the drone could not reach.
Hence, Noflyzone class is defined because the given no-fly-zones json contains four GeoJson polygons and we could store all the information (e.g., a list points which from the polygon coordinates) and methods for checking whether the drone passes the area in this class.

Httprequest. Httprequest class has one static method whose parses argument input are a httplink and a port number, we could get the corresponding String downloaded from the websever.
We need to use this method many times in the application, hence we create this class, and the method is public and static.

Writefile. Writefile class generate and output files. The class has three static methods because we might want to use them in some other classes in the future.
Writefile is a utility class for handling input and output. We might want to use this class in future tasks.

Error. Error class is for checking whether there are input errors when the user input six arguments. If there are exception occur, this class will show user friendly error to let users understand.
Error is a utility class for checking input exceptions. This class is independent of the whole project. Hence it is separated from other classes.

2. Class documentation.

2.1 App Class

Methods:

public static void main (String[] args)

App class only contains a main function which describe the workflow of the whole program. It will initialize three Map object and three GreedyDrone object which use different Greedy algorithms. And output the two generated files.

2.2 Drone Class

Fields:

- **protected static final int maxdegree = 360;**
  The maximum degree which the drone can rotate (360 degree == 0 degree)
- **protected static final int mindegree = 0;**
  The minimum degree which the drone can rotate (0 degree == 360 degree)
- **protected static final int addegree = 10;**
  Drone can only be sent in a direction which is a multiple of ten degrees.
- **protected static final double steplength = 0.0003;**
  One move is a straight line of length 0.0003 degrees.
- **protected static final double readsensordistance = 0.0002;**
  Drone can read within 0.0002 degrees of a air quality sensor.
- **protected static final int maxstep = 150;**
  Maximum step of one flight is 150.

The above fields are made into static since all the objects in Drone class will have the same properties, this will save the memory and make the program runs faster.

- **protected final List<Point> route = new ArrayList<> ();**
  The output of the flight route listed in ordered points.
- **protected final List<Integer> degreelist = new ArrayList<> ();**
  The ordered list which the drone made (each integer from the list is 0-350).
- **protected final List<String> outputwords = new ArrayList<> ();**
  The ordered output all sensors (null for no sensors reading at that step).
- **protected final Point startpoint;** The drone start point.

The above fields are made into final since they are not supposed to be changed anywhere when the program is operating.

- **protected Point nowlocation;** The drone current location.
- **protected int step = 0;** Initialize number of steps to 0(maximum 150).
- **private String flightpath = "";**
  Final output flightPath in String type, initialize to an empty String.
- **private Feature featurepath;** Final output route in Feature type.

Constructor:

- **public Drone (Point startpoint, Map map)**
  This will create a Drone object, and it will initialize both startpoint and nowlocation in the field to startpoint. The rest data will be added by using below methods.

Methods:

- **protected Point getNextLocation (int degree)**

  The drone will use the trigonometric function to calculate the new latitude and longitude by rotating the input degree and go one step toward that degree. Return the next point by using the new latitude and longitude (Will not change the drone current position).
- **protected Point nextLocation (int degree)**

  The drone will use the trigonometric function to calculate the new latitude and longitude by rotating the input degree and go one step toward that degree. Return the next point by using the new latitude and longitude and change the drone current position to the return point.
- **protected boolean checkLocation (Point point, Noflyzone noflyzone)**

  This method is for checking whether the drone could reach that point by a given Noflyzone object. The method will use two methods of the Noflyzone object, it will check whether the point is in the confinement area or in no-fly-zones or the drone have already been this point before. If any one of the three judgments is true, this method will return false.
- **protected double distance (Point A, Point B)**

  This method will calculate and return the Euclidean distance between point A and B.
- **protected int[] checkReadSensor (List<Point> sensorlocation)**

  This method will return an int array which contains whether the drone can read from any sensor (1 for readable and 0 for unreadable), and the input argument is a list of Point contains each sensor location point. The method will check the Euclidean distance between each sensor location and drone current location. If the distance < 0.0002, set 1 in the corresponding place of the array, else set 0.
- **protected void setStringFlightPath ()**

  This method converts the flight Path to String and store it. The method will use a for loop to add String information of each step to the field flightpath.
- **protected void setFeatureFlightPath ()**

  This method converts the route to feature and store it. This method will convert LineString object to a Feature by using Feature.fromGeometry (linestring) and set the field featurepath.
- **protected int getStep ()**
- **protected Feature getFeatureFlightPath ()**
- **protected String getStringFlightPath ()**

  These are the getters for corresponding fields of Drone object.

2.3 GreedyDrone Class

Fields:

- **private final Noflyzone noflyzone;**

  The none fly-zones which the drone could not reach.

Constructor:

- **public GreedyDrone (Point startpoint, Map map, String addorminus)**

This will create a GreedyDrone object, and it will initialize both startpoint and nowlocation in the field to startpoint. And the drone will use Greedy algorithm to find a route on the input argument map. String addorminus is to decide which Greedy algorithm to use. “add” is for when we meet an obstacle, the drone will continue to add degree to bypass it, “minus” is for when we meet an obstacle, the drone will continue to minus degree to bypass it. “first” means we find another degree which we could bypass it.

Methods:

- **private Point getClosestSensor (List<Point> sensorlocation)**

  This method will return the closest sensor from drone current location. Using for loop to calculate the Euclidean distance between each sensor location and drone current location and return the corresponding shortest distance point of sensor.

- **private void updateFlightInfo (int degree)**

  This method will update the fields of route, nowlocation, degreelist after movement by rotating degree.

- **private void greedyOneStep (List<Point> sensorlocation, String addorminus)**

  This is the method of Greedy algorithm for one step and update the following drone status. This will be a detailed introduction in Section 3.

- **private int getNextMinusTurningDegree (int degree)**

  Get the next turning Degree (Minus 10 degree each time to check) for the drone in a reachable area. This will be a detailed introduction in Section 3.

- **private int getNextAddTurningDegree (int degree)**

  Same as getNextMinusTurningDegree () but we Add degree. Get the next turning Degree (Add 10 degree each time to check) for the drone in a reachable area. This will be a detailed introduction in Section 3.

- **public void greedy (List<Point> sensorlocation, List<String> wordslist, String addorminus)**

  This method is the full Greedy algorithm and update the following drone status. The parsed input is a list of point of each sensor location, a list of String contains each corresponding w3words of sensorlocation and a String "first"(normal Greedy) or "add"(add degree when meet a no_fly_drone) or "minus"(minus degree when meet a no_fly_drone). This will be a detailed introduction in Section 3.

2.4 Map Class

Fields:

- **private final String port; private final String year; private final String month; private final String date;**

  Each map has port, year, month, date to identify.

- **private ArrayList<Sensor> sensorlist = new ArrayList<>();**

  A list of sensors which contains each sensor in this map.

- **private final List<Point> locations = new ArrayList<>();**

A list of Points which contains each sensor's location.

- **private final List<String> w3words = new ArrayList<>();**
  A list of Strings which contains each w3words (Type in String).
- **private final List<Double> battery = new ArrayList<>();**
  A list of Double which contains each sensors' battery.
- **private final List<Feature> featurelist = new ArrayList<>();**
  A list of Feature which contains each sensor (type in Feature).
- **private FeatureCollection fc;**
  A FeatureCollection which contains all information of this map.

Constructor:

- **public Map (String port, String year, String month, String date)**
  This will create a Map object, and it will initialize port, year, month and date in the field. And automatically generate the whole map which contains every sensors information.

Methods:

- **public List<Point> getLocations ()**
- **public List<String> getW3words ()**
- **public List<Feature> getFeatureList ()**
- **public FeatureCollection getFeatureCollection ()**
- **public String getPort ()** **public String getYear ()**
- **public String getMonth ()** **public String getDate ()**
  These are the getters for corresponding fields of Map object.

2.5 Sensor Class

Fields:

- **(public) String location;** The location of the sensor (in w3words form).
- **(public) Double battery;** Battery of the sensor.
- **(public) String reading;** The air quality reading of the sensor.
  The above three fields are parsed by the w3word details.json on the webserver.
- **public String rgbstring;** The color of sensor shown on the GeoJson graph.
- **public String markersymbol;**
  The marker symbol of sensor shown on the GeoJson graph.
- **public Point point;** The coordinates of the sensor.
- **private String port;** The port number.

Methods:

- **public void setLatLng ()**
  This method will set the coordinates of the sensor from its corresponding w3words. It will parse the corresponding w3word and get the coordinates of the sensor in this word area. Then set the Point field to this coordinate.
- **public void setRGBstring ()**
  This method will set the RGBstring and the corresponding Marker symbol. This is for creating the GeoJson point object in the future. It is mainly

hardcoding to decide the RGBstring and the corresponding Marker symbol by the readings and the battery.

- **public Feature toFeature ()**

  This method will firstly add four properties (location, rgb-string, marker color, marker-symbol) to a sensor point in order to create the GeoJson point object in the future and secondly return a feature of the sensor which contains four properties.

- **public void setPort(String port)**

  A setter of setting port number.

2.6 W3Word Class

Fields:

- **(public) String country;** Country of that w3words.
- **(public) String nearestPlace;** "Edinburgh" in this task.
- **(public) Square square;**

  The coordinates of the south_west & north_east of the square.
- **(public) Coordinate coordinates;** Contains the latitude & longitude.
- **(public) String words;** The w3word (e.g. xxxx.xxxx.xxxx).
- **(public) String language;** "en" in this task.
- **(public) String map;** The corresponding http String.

  The above seven fields are parsed by the w3word details.json on the webserver.

2.7 Square Class

Fields:

- **(public) Coordinate southwest;**

  The coordinates of the south_west of the square.
- **(public) Coordinate northeast;**

  The coordinates of the north_east of the square.

  The above two fields are parsed by the w3word details.json on the webserver.

2.8 Coordinate Class

Fields:

- **(public) Double lng; (public) Double lat;**

  A Coordinate class contains the latitude and the longitude of a point and the above two fields are parsed by the w3word details.json on the webserver.

2.9 Noflyzone Class

Fields:

- **private final double[] lat1; private final double[] lng1; private final double[] lat2; private final double[] lng2; private final double[] lat3;**

private final double[] lng3; private final double[] lat4; private final double[] lng4;

Each array above contains the number of polygon latitudes or longitudes.

- **private static final double confinment_area_lat1 = 55.942617;**
- **private static final double confinment_area_lng1 = -3.192473;**

  The north-west corner of the confinement area: Top of the Meadows (55.946233, −3.192473)
- **private static final double confinment_area_lat2 = 55.946233;**
- **private static final double confinment_area_lng2 = -3.184319;**

  The south-west corner of the confinement area: KFC (55.946233, −3.184319)

Constructor:

- **public Noflyzone (String port)**

  The constructor takes a port number as input and then using GeoJson type casting breaks a GeoJson feature to four polygons and store latitudes and longitudes of each polygon in above fields.

Methods:

- **private double[] getArray (List<Point> list, String type)**

  This method will sperate a list of point into an array of double, either contains each latitude or longitude of a point in the list. input arguments is a list of Point (A list of Point from a polygon in this task) and a String either "lat" or "lng", "lat" for getting all latitude and "lng" for getting all longitude. Then return an array of double which contains either latitude or longitude from the Point list.
- **public boolean notInConfinement (Point point)**

  This method will check whether the point is in the confinement area. The input arguments are a point which we want to know whether is in the confinement area. Return a Boolean value (true for the point is not in the confinement area).
- **private boolean isIntersect (double lat1, double lng1, double lat2, double lng2, double lat3, double lng3, double lat4, double lng4)**

  This method checks whether the first line(l1(lat1,lng1)-l2(lat2,lng2)) is intersect with the second line(l3(lat3,lng3)-l4(lat4,lng4)). This method use pure Math to calculate.
- **private boolean notIntersectOnePolygon (Point start, Point end, double[] polygonPoint_lat, double[] polygonPoint_lng)**

  This method will check whether a line is intersecting with one of the lines of one polygon. The input arguments are the start point of the line segment and the end point of the line segment and a double array contains all the latitude of the points from the polygon and a double array contains all the longitude of the points from the polygon. And finally return a Boolean value (true for the line is not intersect with the lines of the polygon).
- **public boolean notInNoFlyZones (Point start, Point end)**

This method will check whether a line is intersecting with one of the lines of four polygons which consist a NoFlyZone. The input arguments are the start point of the checking line segment and the end point of the checking line segment. Return a Boolean value (true for the line is not intersect with the lines of all four polygons which consist the NoFlyZone object).

2.10 Httprequest Class

Methods:

- **public static String getResponce (String urlstring, String port)**

  This method will download the information as a string from a GeoJson file with the specified url String and we need to use this method many times in the app, hence it is a public static method. The input parametres are a urlstring which is the string of the http link and a port which we will Run web server on this port number. Return Everything included in the specific GeoJson file as a string. We throw IOException which is an error line which could let users easily understand if we cannot download anything and exit the application.

2.11 Writefile Class

Methods:

- **private static void writeGeojsonFile (String geojson, String date, String month, String year)**

  This method will write the output GeoJson file in the current working directory. The input arguments are a String of GeoJson which is a map in String format which contains 33markers and the drone flight path and a corresponding date (date, month, year) of this map. If fail to write a file, it throws an exception of a single line error to let users understand what happen.

- **private static void writeTxtFile (String flightpath, String date, String month, String year)**

  This method will write the output Txt file in the current working directory. The input arguments are a String of flightpath which is is the drone flight path in String type and a corresponding date (date, month, year) of this map. If fail to write a file, it throws an exception of a single line error to let users understand what happen.

- **public static void outputTwoFiles (GreedyDrone drone, Map map)**

  This method will write both geojson file and txt file in the current working directory. The input parameters are a GreedyDrone drone which is the drone which we use and a map which is the corresponding map which the drone flight.

2.12 Error Class

Methods:

- **public static void showUsersError (String date, String month, String year, String latitude, String longitude, String randomseed, String port)**

This a static method for checking whether there are errors caused by input arguments and let users understand the error. The method will check if the date or month or year cannot convert to integer, print an exception to let users know.

3. Drone control algorithm.

This section covers up the algorithm part of the program. The classes correspond with this part are **Drone.java**, **GreedyDrone.java** and **Noflyzone.java**. Drone control algorithm contains two parts, one is for the Greedy algorithm which shows the basic movements and direction decisions made by the drone. And the second part is about the decisions made by the drone when it meet obstacles or the confinement areas.

3.1. Greedy algorithm

Introduction of the Greedy algorithm: A greedy algorithm is a simple, intuitive algorithm which is used in optimization problems. The algorithm makes the optimal choice at each step as it attempts to find the overall optimal way to solve the entire problem.

3.1.1 Greedy algorithm Procedure:

Step 1. Calculating the Euclidean distances between the drone current location and each sensor location from the sensor list which need to be read. Find the corresponding sensor with the shortest distance.

Step 2. Calculating 36 next positions of the drone rotation every 10 degrees from its current location. Calculating the Euclidean distances between each of these positions and the sensor which we get in *Step 1*. Find the corresponding degree with the shortest distance.

Step 3. Drone will move one step in this degree which calculated in *Step 2*. Drone will update its current location.

Step 4. Checking whether drone can read the sensor which we get in *Step 1* (distance between the drone current location and the sensor should within 0.0002 in order to read). There will be two options below,

- Option A: Readable. (distance within 0.0002)
  Drone will read and store the information provided by the sensor and delete this sensor from the sensor list which need to be read. If the list is empty now, then end the procedure else go back to *Step 1*.
- Option B: Unreadable. (distance >= 0.0002)
  Go back to *Step 1*.

3.1.2 Go back to start point algorithm Procedure:

After the drone have read and store all the sensors information, it needs to go back to the start point. Hence, we add the drone start point to the sensor list which need to be read and continue to do Greedy Drone algorithm Procedure from step

1. When drone end the Greedy procedure, it will be close to its original start point within 0.0002. (Noted: we only add start point to the sensor list once in the whole procedure otherwise will add the start point forever and never end the program).

```
public void greedy(List<Point> sensorlocation, List<String> wordslist, String addorminus) {
    route.add(startpoint); // First add the start point to the route.
    int doOnce = 0; // This variable is related to the drone returning to the starting point.
    // After read all sensors and return to the start point or the step is over 150,
    // we stop the drone.
    while (sensorlocation.size() > 0 && doOnce < 2 && step < maxstep) {
```

Figure 2: segment of greedy function

In Figure 2, there is a defined integer called doOnce and it will be added only when there is only one sensor left in the sensor list. When doOnce >=2, the Greedy algorithm procedure will be stopped. Hence, it will make sure that add only once start point to the sensor list.

3.2. Confinement area & Obstacle Avoiding Mechanism

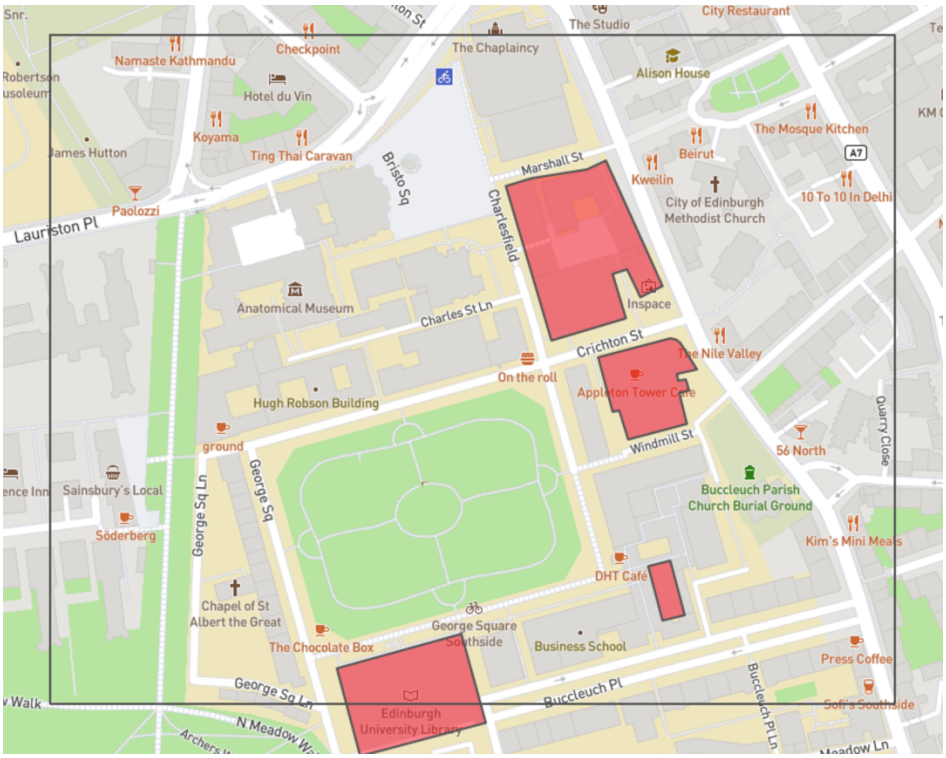

Figure 3: Confinement area & no-fly-zones

This part contains the mechanism of both confinement area and the no-fly-zone which consists with four obstacles. In order to avoid these no-fly-zones, drone will apply two mechanisms below.

3.2.1 Confinement area

In Figure 3, area inside the square in black is the drone confinement area. Drone could only fly in the confinement area.

Check whether the drone is in the confinement area Procedure:

Step 1. Check whether the latitude of point we want to check is between the latitude of the left-bottom point of the confinement area and the latitude of the right-up point of the confinement area. And also check whether the longitude of point we want to check is between the longitude of the left-bottom point of the confinement area and the longitude of the right-up point of the confinement area.

Step 2. If both result in *Step 1* are true, return true else false.

3.2.2 No-fly-zones

In Figure 3, those red polygons are the no-fly-zones. Testing whether the drone passed the confinement area using the same method of checking point is not enough. Right Figure 4 is a counter example. Therefore, we need to check whether the flight path intersect with the edges of the four polygons. If the flight path intersects with any edges of the point, which means the drone could not flight though this path and need to find another path.

Figure 4: forbidden flight path

3.2.3 Avoid obstacles mechanism apply on Greedy algorithm

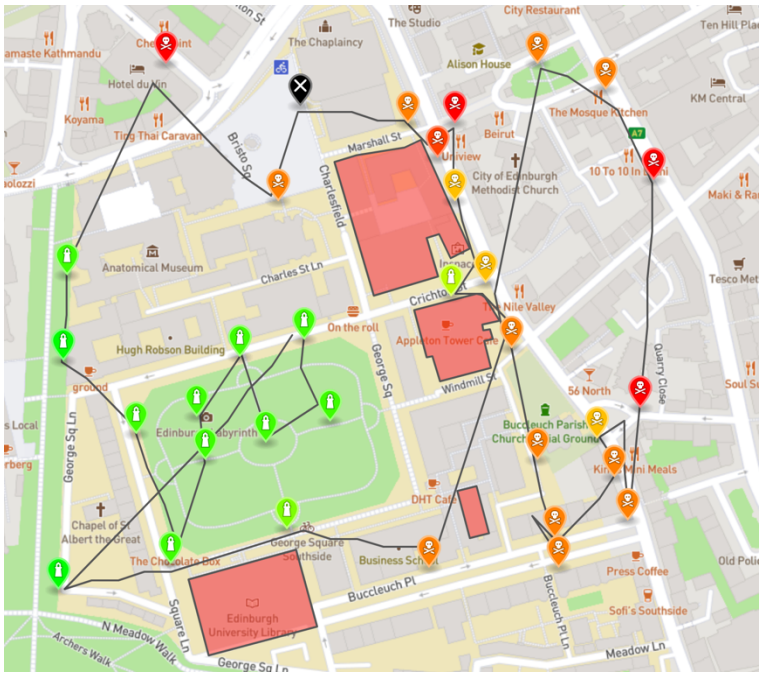

Figure 5: 01-01-2020 graph (step: 104)

This algorithm will let the drone fly almost along the edge by clockwise (add degree) or anti-clockwise (minus degree) while meet an obstacle.

Avoid obstacles mechanism apply on Greedy algorithm Procedure:

Step 1. Calculating the Euclidean distances between the drone current location and each sensor location from the sensor list which need to be read. Find the corresponding sensor with the shortest distance.

Step 2. Calculating 36 next positions of the drone rotation every 10 degrees from its current location. Calculating the Euclidean distances between each of these positions and the sensor which we get in *Step 1*. Find the corresponding degree with the shortest distance.

Step 3. We need to check whether the drone can move one step in this degree. If the drone pass no-fly-zones or not in the confinement area, we need to change path. Below are two options.

- Option A: Drone can move to that point.
  Drone will move one step in the direction of degree which calculated in *Step 1.*
- Option B: Drone cannot move to that point.
  Add 10 to the degree which calculated in *Step 1.* And go to *Step 3.* (Noted: there are another algorithm for continuous to minus 10 degree.)

Step 4. Checking whether drone can read the sensor which we get in *Step 1* (distance between the drone current location and the sensor should within 0.0002 in order to read). There will be two options below,

- Option A: Readable. (distance within 0.0002)
  Drone will read and store the information provided by the sensor and delete this sensor from the sensor list which need to be read. If the list is empty now, then end the procedure else go back to *Step 1*.
- Option B: Unreadable. (distance >= 0.0002)
  Go back to *Step 1*.

For example, below are two graphs using the same map but different rotate direction when meet an obstacle. They will bypass from different sides of the buildings. The Figure 6 is 04-04-2020 using add degree (Step:113). And Figure 7 is 04-04-2020 using minus degree (Step:105). We will choose the path which have the smallest number of steps.

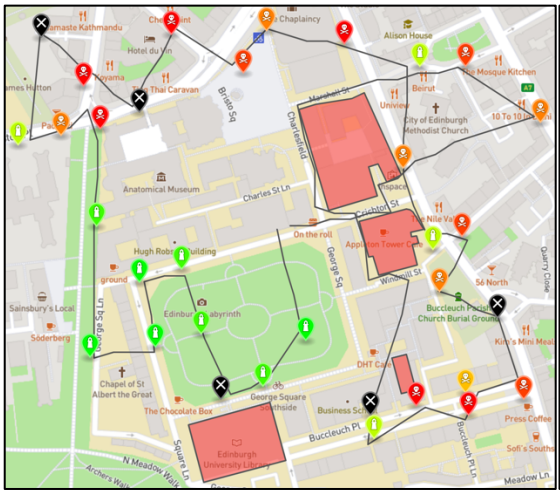

Figure 6 (Step:113)

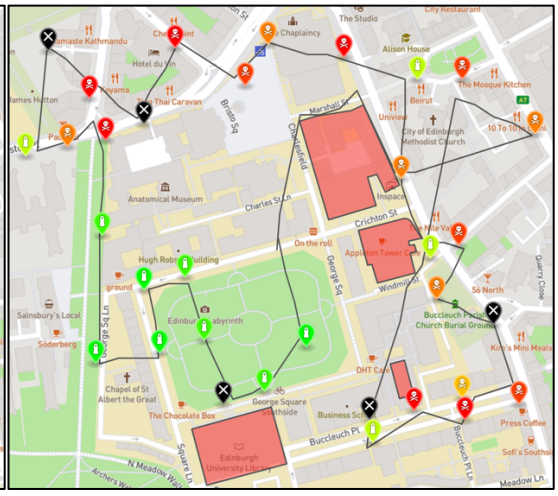

Figure 7 (Step:105)