INFORMATICS LARGE PRACTICAL

PROJECT REPORT

WINI LAU - S1846175

DECEMBER 2020

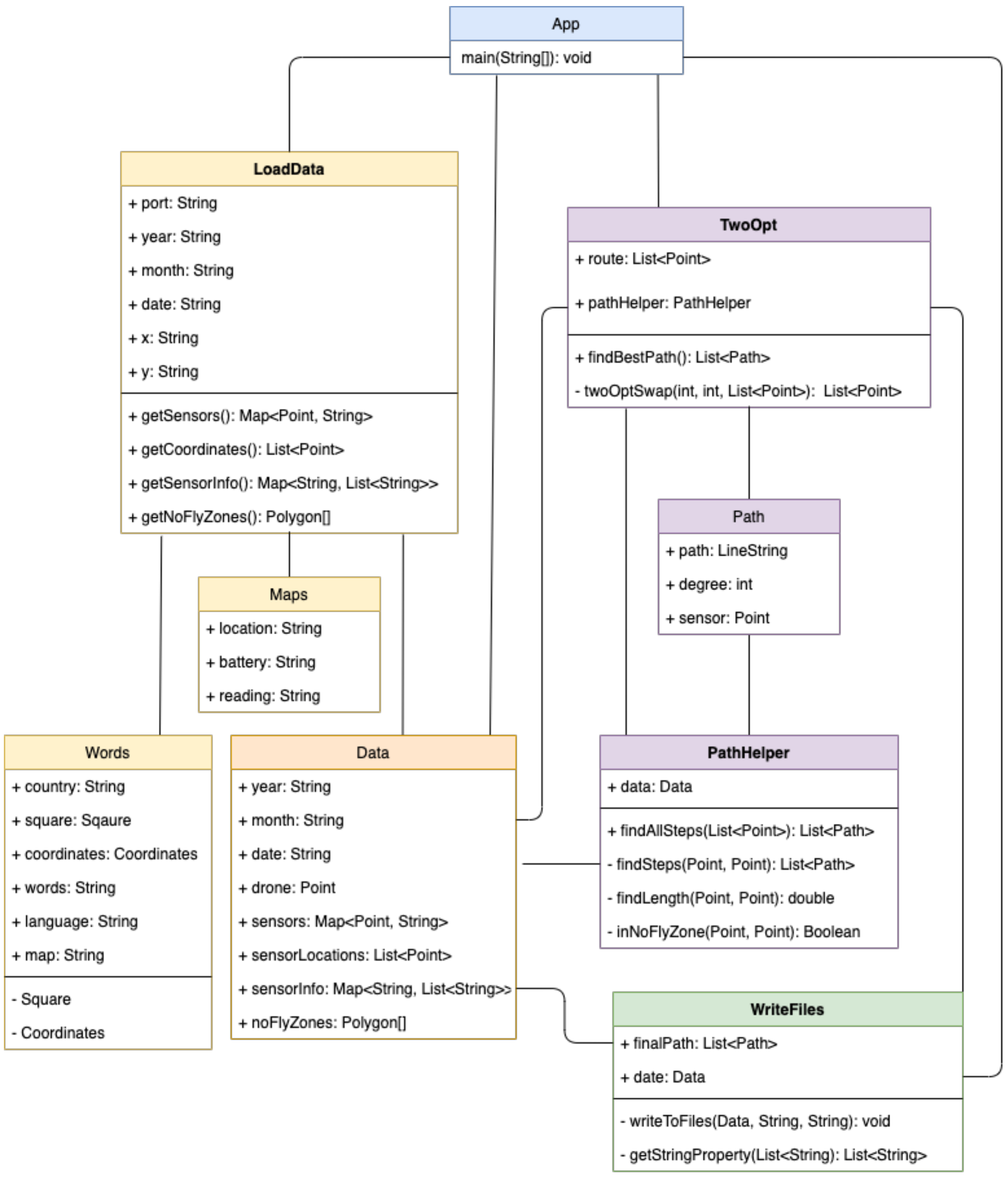
Course by:

Stephen Gilmore and Paul Jackson, University of Edinburgh School of Informatics

Table of Contents

PART 1: SOFTWARE ARCHITECTURE DESCRIPTION

Figure 1.1: class diagram


App
main(String[]): void
LoadData
+ port: String
+ year: String
+ month: String
+ date: String
+ x: String
+ y: String
+ getSensors(): Map<Point, String>
+ getCoordinates(): List<Point>
+ getSensorInfo(): Map<String, List<String>>
+ getNoFlyZones(): Polygon[]
TwoOpt
+ route: List<Point>
+ pathHelper: PathHelper
+ findBestPath(): List<Path>
- twoOptSwap(int, int, List<Point>): List<Point>
Path
+ path: LineString
+ degree: int
+ sensor: Point
Maps
+ location: String
+ battery: String
+ reading: String
Words
+ country: String
+ square: Sqaure
+ coordinates: Coordinates
+ words: String
+ language: String
+ map: String
- Square
- Coordinates
Data
+ year: String
+ month: String
+ date: String
+ drone: Point
+ sensors: Map<Point, String>
+ sensorLocations: List<Point>
+ sensorInfo: Map<String, List<String>>
+ noFlyZones: Polygon[]
PathHelper
+ data: Data
+ findAllSteps(List<Point>): List<Path>
- findSteps(Point, Point): List<Path>
- findLength(Point, Point): double
- inNoFlyZone(Point, Point): Boolean
WriteFiles
+ finalPath: List<Path>
+ date: Data
- writeToFiles(Data, String, String): void
- getStringProperty(List<String>): List<String>

To better understand the tasks I needed to do, I broke down the entire coursework into three basic steps:

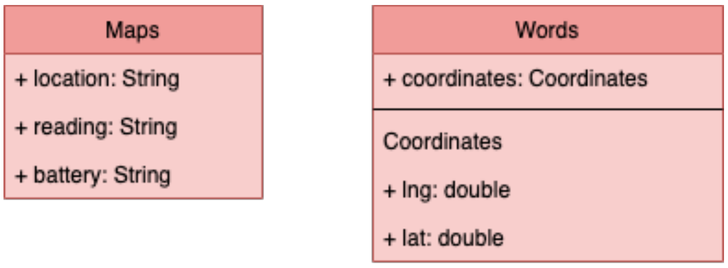
1. Load data from webserver
2. Compute path
3. Create output files

Then I structured my code based on these 3 steps:

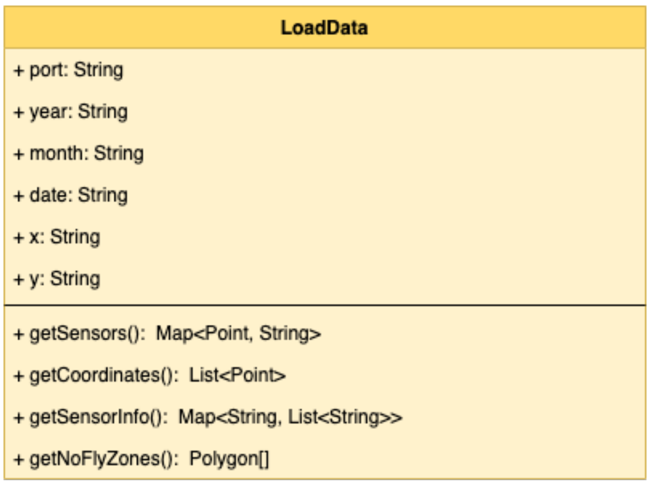
Figure 1.1 is a class diagram I made to represent the class structure of my function. To explain simply, I separated the function into four categories: the first one being LoadData, Words, and Maps (coloured yellow). The class LoadData is used for loading the data from the web server and storing the data into Lists, Maps, and Arrays depending on future usage. The Words and Maps class are needed because they are used to obtain data from the server. The next category is the Data class (coloured orange). I created this class to store the data from calling the methods of the LoadData class, so I don't have to make multiple calls to the LoadData methods for efficiency. Next category consists of TwoOpt, Path, and PathHelper (coloured purple). These classes all contribute to finding the final path of the drone in which PathHelper has helper methods to help the TwoOpt class sort the route using the two opt heuristic. The Path class is used to store the information for each step the drone takes including the LineString, the degree, and a sensor if it has reached any. Lastly, in the last category is my WriteFiles class (coloured green). This class computes the content of each of the output files from the loaded data and the final path obtained from previous code. Then it creates and writes the two output files by using the FileWriter function from Java.

PART 2: CLASS DOCUMENTATION

2.1 Maps, Words, and LoadData



These are the two classes required to access data from their corresponding folder on the webserver. The Maps class has three fields corresponding the data from the maps folder: the location as a what3words String. and the reading and battery as Strings. The Words class actually had a lot more fields but since the only part that matters from the words folder is the Coordinates, I decided to only show that here. (see figure 1.1 for full class diagram).


LoadData
+ port: String
+ year: String
+ month: String
+ date: String
+ x: String
+ y: String
+ getSensors(): Map<Point, String>
+ getCoordinates(): List<Point>
+ getSensorInfo(): Map<String, List<String>>
+ getNoFlyZones(): Polygon[]

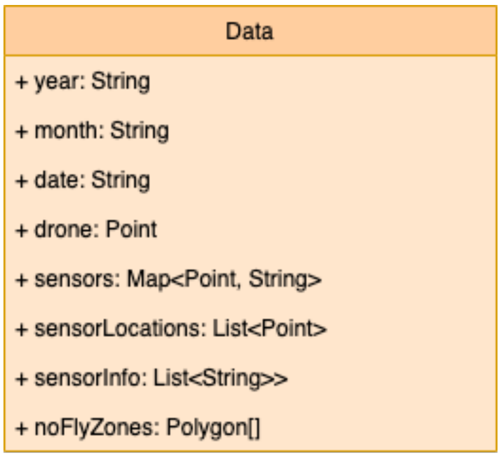
LoadData takes the required arguments from main and accesses the data from the corresponding folder(s) in the webserver.

- The first method getSensors() returns a Map of sensor location as a Point to the sensor location as a what3word. This is needed for the output file, namely the

WriteFiles class. To get the return values, I first get a response from the maps folder of the corresponding date and convert it to an ArrayList of Maps. I then loop through ArrayList and get to location value of each sensor which is a what3word String. For each iteration, I access the words folder with the what3word and convert the data into the Words class. Finally, I covert the latitude and longitude accessed from the coordinates property of the words folder into a Point and put the Point into a Map with the what3word String. This process repeats until all of the ArrayList of Maps from earlier is reached. (33 for the examples give in the coursework).

- The second method getCoordinates() is similar to the first method but it only returns the sensor locations as a List of Points. This is used for computing the path namely the PathHelper class and the TwoOpt class. The process is identical as the first method but instead of putting the Point and the what3word into a Map, this method puts the Points into a List and discards the what3word.
- The third method getSensorInfo() returns a Map of the sensor as what3words to a List of String of the reading and battery values of the sensors. This is needed for the output geojson file. To get the values I only accessed the Maps folder this time. Then I converted the data to an ArrayList of Maps. I looped through the ArrayList and for each iteration I put the location as key and the List of the reading and the battery as a value of the map.
- The last method getNoFlyZones() is used to access the buildings folder and get the no fly zones as an Array of Polygons. Similar to all the methods before, I just accessed the data of the buildings folder, and since the data is already a GeoJson String I didn't need to make a class for it. I simply converted the file into a List of Feature and then get the polygons for each Feature.

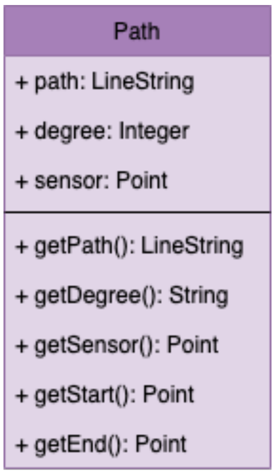
2.2 Data


Data
+ year: String
+ month: String
+ date: String
+ drone: Point
+ sensors: Map<Point, String>
+ sensorLocations: List<Point>
+ sensorInfo: List<String>>
+ noFlyZones: Polygon[]

I created this class to store information I need throughout the entire program for easy access. The stored values are:

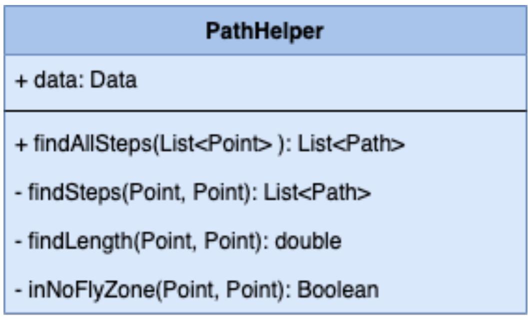
- Year, month, date – need for creating file name
- Drone – starting point of drone, obtained from the main arguments (the latitude and longitude); need for computing path
- Sensors – the location of sensors as a Map of Points to what3words; need for both output files
- sensorLocations – the location of sensors as a List of Points; need for algorithm
- sensorInfo - a map of the what3words of sensors to the air quality reading and battery values.
- noFlyZones – an Array of Polygons that the drone can't fly into; need for algorithm

2.3 Path



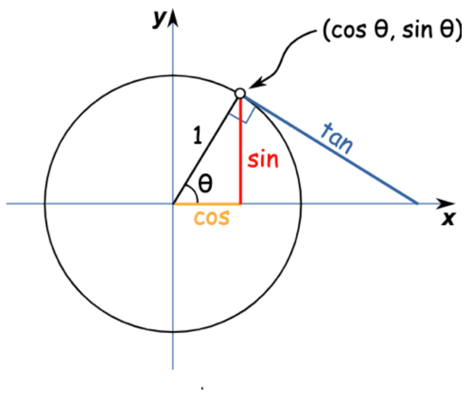
This class is what I used to store the steps of a path. It stores three values: the LineString of the step (always 0.0003 in length), the degree (from 0 to 350 in increments of 10) of the corresponding step, and the sensor this step reached if any (otherwise null). Creating this class makes it easier for me because I don't have to worry about storing and access the three different values when they can just all be accessed in a class. Apart for the getters, this class also has two method that gets the starting point and ending point of the step which makes the code cleaner.

2.4 PathHelper


PathHelper
+ data: Data
+ findAllSteps(List<Point>): List<Path>
- findSteps(Point, Point): List<Path>
- findLength(Point, Point): double
- inNoFlyZone(Point, Point): Boolean

The PathHelper class only has one public method, findAllSteps(), the others are helper methods. findAllSteps() is used by TwoOpt to find all the steps of a specific route (List<Point>).

To obtain that, I first have to gather the 36 different moves the drone could go. I did this in the constructor by using the unit circle formula:



I looped through the 36 degrees and applied this formula (x = cos(θ) and y = sin(θ)) on each one then I multiplied the results by 0.0003. I store these values in a 36 by 2 2D array where column 0 represents the x values and column 1 represents the y values. These are the 36 options in which my drone can move in each step.

To explain how I implemented this method I will first explain the helper methods:

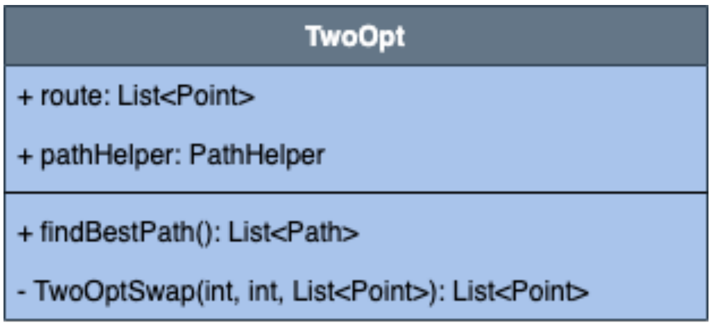
- findLength(Point, Point) takes two points and find the Euclidean distance by getting the differences in the x's and y's or the points and then getting the hypotenuse. It is a significant part of the findSteps(Point, Point) method.
- inNoflyZone(Point, Point) checks if the LineString formed by the input points is in the no fly zones and the if it exceeds the outer boundary. For this method, I got all the LineStrings that forms the no fly zones Polygons and I store them in a List of LineString called noFlyStrings. Then I add the four lines that makes up the outer boundary to that list. Lastly, I loop through the list and use the java package Line2D.linesIntersect to check if the input LineString(derived from the input points) intersects with any of the noFlyStrings, if it does then return true. After the loop return false because it is not in the no fly zones.
- findSteps(Point start, Point end) returns a list of Path that represents all the steps it takes to get from the starting point to the end point. I first initialise a Point called current as start to keep track of the drone's current location. Then I initialised a Point called best representing the best option the drone could take and I set it as

the first option (i.e. start.longitude() + directions[0][0] and start.latitude() + directions[0][1]). I also initialise an int degree to 0 because the best is at 0 degrees right now. Then I check by using findLength and inNoFlyZone if best already satisfies the requirement that the distance between best and end is less than 0.0002 and the line from start to best is not in the no fly zones. If so, I add make a Path with the LineString from start to best, I add the degree zero and the sensor end then I add it to an empty List and return. However, if the requirements were not met at degree 0, while the distance between best and end is greater than or equals to 0.0002 or the line of current to best is in no fly zone:

- I loop through all the 36 options and I find all the ones that are not in the noFlyZones and add them to a List of Points called valid while storing their corresponding degrees in a List of Integer called degrees.
- Then I initialised best as valid.get(0), and I loop through the rest to find one that has a shorter distance to the end. If I find one, I set best to the new Point.
- After that I make a new Path with LineString of current to best, and corresponding degree from degrees, and end as sensor if distance between best and end is less than 0.0002.
- Lastly, I set current to best and the process repeats until the sensor is within range.

After implementing all that, the last step I did was to create the method findAllSteps(List<Point>) in which it loops through the input List of Points and apply findSteps on the first point to the next, append the results to the list, then call findSteps but set the start to the end of the last path and end and the end to the next point in the input route, this process repeats until the last Point of the list is reached.

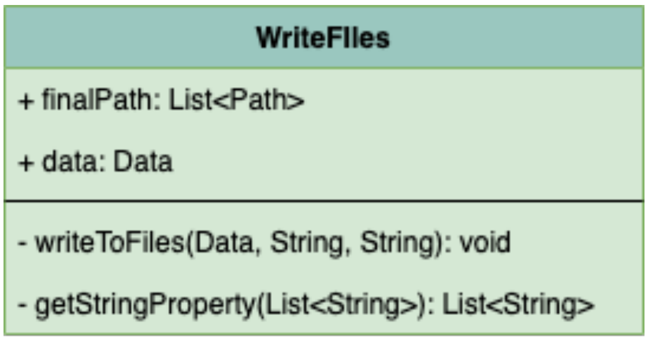
2.5 TwoOpt


TwoOpt
+ route: List<Point>
+ pathHelper: PathHelper
+ findBestPath(): List<Path>
- TwoOptSwap(int, int, List<Point>): List<Point>

This is the class where I compute the best estimated path for the drone. I used a two-opt algorithm because I remember learning it from last year and how it performs better than greedy on average. There are two methods in this class:

- TwoOptSwap(int, int, List<Point>) is a helper method that takes in an integer i, an integer k and a List<Point> route; it out puts a different route with the that has the elements from i to k reversed from the original. To implement this I added elements from route.get(0) to from route.get(i-1) to a new List of Point, then I add the reversed middle part by using the Collections.reverse java package and lastly I the elements from k+1 to the size of the original size – 1.
- In the main method findBestPath() , I use the 2 opt algorithm to find best path. I set the best route to the initial route, then I loop through all possible i to k within size and then compare if the swapped one has lower number of steps. if so, set best to new route. repeat this process until improve == 2 is reached. Since the only different part of the route is the reversed part, I only compared the steps it takes to do the route from i-1 to k+1. And I find the path by calling findAllSteps in PathHelper.

2.6 WriteFiles


WriteFIles
+ finalPath: List<Path>
+ data: Data
- writeToFiles(Data, String, String): void
- getStringProperty(List<String>): List<String>

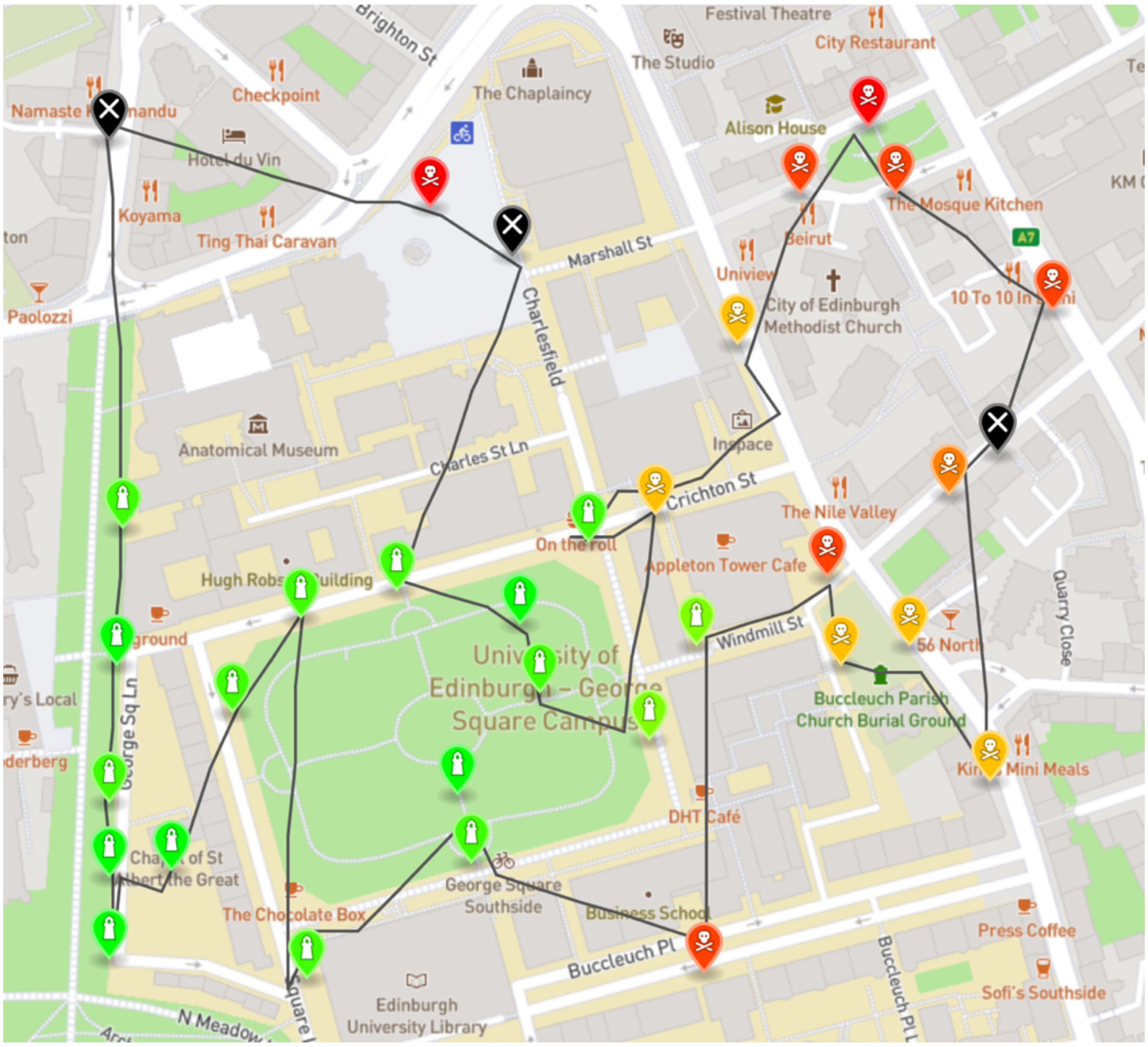
This class is used for creating and writing the geojson and text output files. Unlike the other classes, because this is the last step of the function and there is no need to store anything, instead of having a main method I just put the code in the constructor. To compute the two files first I needed to make 2 Strings, and since it was going to be derived from the same output, the final path, I did it together.

- To get the flightpath String, I looped through the final path (an input to the constructor) which is a List of Path obtained from the findBestPath method in the TwoOpt class. For each of the Path I append a line of "i +1"(the iterator), the starting point as longitudes and latitudes, the degree, the end point as longitudes and latitudes, and I call data.sensors to find the corresponding what3word of the current sensor if any. (and end of line)
- To get the geojson String, in the same loop as above, first add the end point of the path to a List a point (with the starting point of drone already in it) create a one LineString representing the path as format suggested. Then check if sensor isn't null, if so, add the Point of the sensor and access data from data.sensorInfo to access reading value and battery level. After getting the data, check which range it is it and what String properties to put with the private helper method getStringProperty(List<String>), then add the string properties to the point.
- After looping through the entire path, I call the writeToFile helper method to create the two files in working directory and write the strings we obtain in each file

PART 3: DRONE CONTROL ALGORITHM

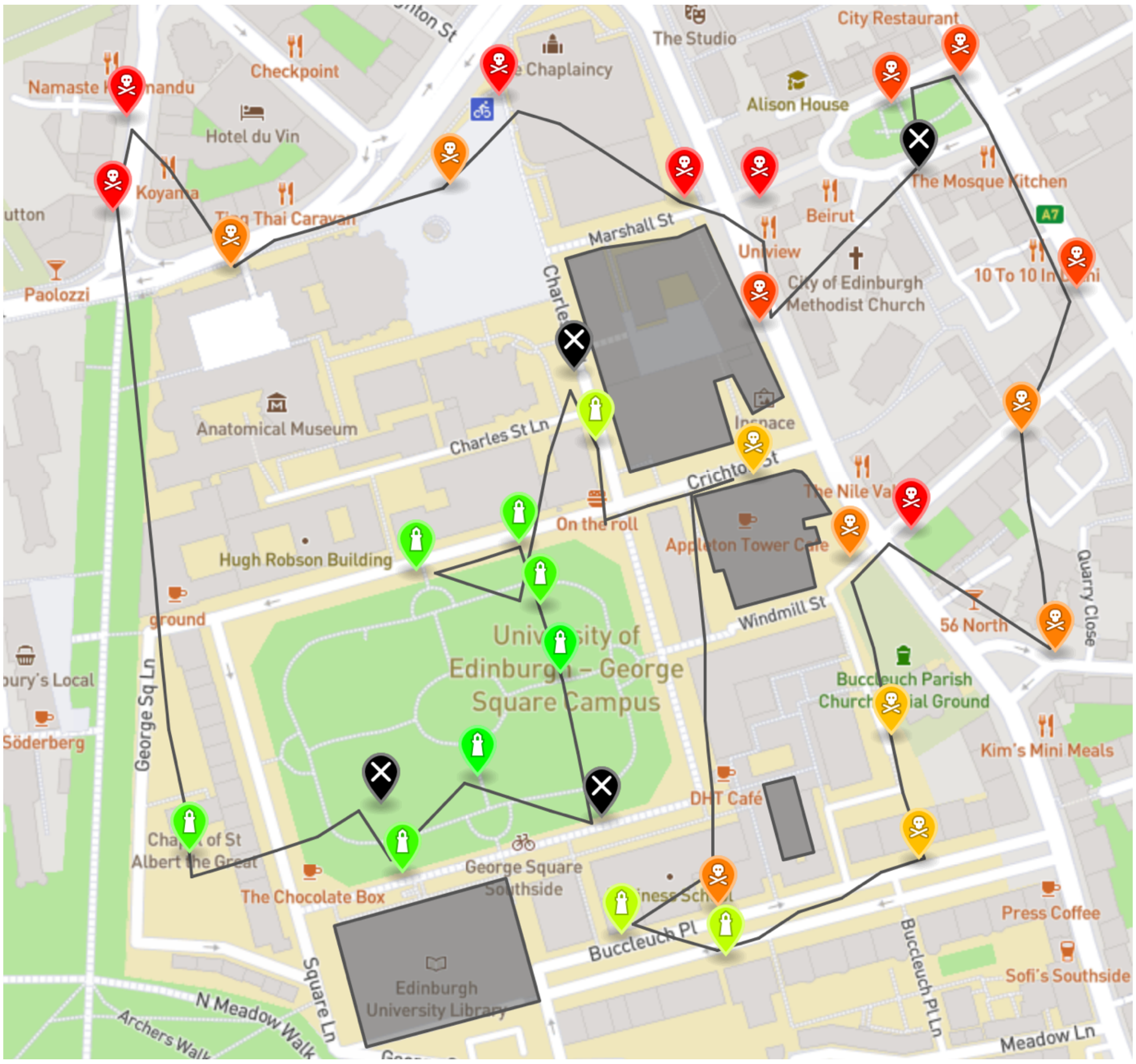
The 2-opt algorithm is one of the most well know algorithm for the Travelling Salesman problem alongside with the greedy search. It takes a random route and improves it by iterating through it and swapping everything between 2 iterating integers. Although the 2-opt algorithm seem to work well and yields decent results, one downfall is its speed. Because of the number of loops 2-opt has to go through in order to yield a good result, it is not realistic to use an algorithm like this for a big data size. This is also one of the reasons why my code is also very slow compared to the time a greedy search would take. My efficiency was also affected because I find the steps first before finding the path. I wanted to do it this way because I think that since the drone can only move in steps instead of going any distance and any direction, finding a path first based on distance would not be the most effitive. However, although my algorithm takes a while to compute, it performs than my old greedy search algorithm especially for cases where there are one or two sensors very far away from all the other sensors and cannot be reached by greed until the end. For example:

Figure 3.1: 12-12-2020.geojson



The sensor at the top left corner would have been the second last to be reached in a greedy search and it would take a lot of steps if the previous sensor is too far away from it. However, with 2-opt search, this is no longer a problem because 2-opt takes into account the whole route instead of just the current distance.

Figure 3.2: 08-04-2020.geojson



Another example is this map. Because there is a big gap for the sensors on the left-hand side, it wouldn't be very efficient to yield a greedy result.

Although it takes a while, I still think the 2-opt is a pretty good algorithm for determining the best path or something because it takes into account the whole route and compares a lot of routes before finding one that is really good.