Informatics Large Practical

Implementation report

S1980526

Contents

1. Software architecture description

In this section I will provide a detailed description of the architecture of my application. Furthermore, I will justify my design choices and explain the reasoning behind my different Java classes.

1.1 Justification

I modelled my program using 16 classes. This seems like many for some, but here I'll outline the reasoning behind it and how it improves encapsulation. I tried to follow the SOLID principle as closely as I was able to.

First, we must request two types of data from the server, obstacles and sensors. I have two classes modelling that, **ObstacleService** and **SensorService**, as the way to retrieve the data from the server differs. (single responsibility) Still, there are some common components of the data retrieval, hence they inherit from the **BackEndService** to retrieve those.

Secondly, we need to log two different types of output. The same reasoning applies; hence I created a **FlightPathLogger** and **ReadingLogger**, which log the flight path (as text file) as well as the geoJSON file. To give a common interface for those loggers, they inherit from an abstract base class, **DroneLogger**, which defines the abstract **log** and **close** methods. I decided to use an abstract class over an interface as the constructor of both subclasses is the same, and they both have two common attributes, the position of the drone as well as the file to which to write.

We also need to define an entry point for our application, the **App** class. It hooks up all the dependencies between the classes.

There are two "real" types in the environment, the **Drone** and **Sensor**s. Therefore, I found it natural to create a class for each of them. To model the position of those, we inherit from the **JTS** coordinate class.

When doing moves, we need some way of validating those, as we must stay within the navigation area. Therefore, I created a class representing the valid navigation area, the **Map** class. Given two coordinates, it can tell us whether we will move outside the navigation area or not. It would have also been possible to model the map only as a JTS **Polygon,** but this class has no methods to integrate obstacles into its boundaries if they are only partially inside the polygon.

We need some way of resolving an air pollution level to a RGB string and a marker symbol, as we need this to visualize each marker. Therefore, I created an Enum, **MarkerProperties**, which maps air pollution levels to the corresponding marker properties.

As we need to return the angles between coordinates and sometimes need to do other calculations with angles, I created a static class, **Angles**, which holds all operations involving angles. This reduces redundancy while abstracting the tricky math away.

Now I'll talk about the classes related to my algorithm. Those are in total four. As my algorithm is a bit more complex, I felt like there was a big need to define different classes, so it is still easy to understand.

The **RoutePlanner** is the "brain" of the algorithm, hooking up each of the other classes related to my algorithm to return the final route. It serves as an abstraction layer over the whole algorithm, providing an easy-to-use interface for the **Drone** class.

The next one would be the **PathFinder**, which finds the shortest paths between different coordinates. It implements the Floyd-Warshall algorithm to do so. As this algorithm can be complex on its own, I wanted to put it in its own class. Furthermore, we

again follow the single responsibility principle, calculating only the shortest paths. It furthermore updates the **VisibilityGraph** distance matrix when shorter distances are found between two nodes.

The **VisibilityGraph** is used to represent the underlying edge distances between the sensors and obstacles. As those calculations involve several checks, it required its own class. Additionally, we use this class as the data container for the distance matrix between all vertices. (obstacle coordinates plus sensors) It also uses the **Graph** interface, which only defines methods to set and get coordinates as well as distances as well as the size of the graph. This is useful as it furthermore abstracts the fact that we can add new coordinates.

The **RouteFinder** is responsible for the "global" optimization of the route visiting all sensors by employing a TSP heuristic. It makes heavy uses of the GraphHopper library. As I tried to keep the use of this library at a bare minimum, it also acts as a layer of abstraction above it.

1.2 Choice of additional libraries

I am making use of two additional libraries which were not named in the coursework specification. The first one of those would be JTS. JTS is a geometry library which provides many useful functionalities to handle geometric shapes. For example, you can easily check if a line or coordinate is contained in a polygon or compute the intersection of two geometries. I make excessive use of this library and some of my classes (Sensor, Map) naturally extend classes in the library.

Furthermore, to get an approximate best route visiting all waypoints, I use the GraphHopper Jsprit library. The reason for that is that I did not want to spend time tuning my algorithm when there are already proven ones out there.

2. Class documentation

2.1 App.java

This class initializes and injects most dependencies. It has the following methods and attributes:

- A private static String variable **URL**, which points to the base url localhost.
- A private static Random variable **random**, which stores the random object initialized with the parsed seed.

Furthermore, the following methods:

- The public static **getRandom** method, which returns the **random** attribute.
- The public static **void main(String[] args)** method. It first parses the input arguments, then creates a Random object (and assigns it to **random**), then initializes the **SensorService** and uses it to pass a List of Sensors into the constructors of the **ReadingLogger**. After that, we construct a **FlightPathLogger.** Then we initialize the **ObstacleService,** pass the obstacles returned by it into the **Map** class, and use the resulting object for the construction of the **VisibilityGraph**. Finally, we initialize the **RoutePlanner,** and inject the necessary objects into the **Drone** class. After that, we start the program by calling **visitSensors()** of the **Drone**.

2.2 Angles.java

The Angles class contains the following collection of methods:

- Public static **calculateAngle(Coordinate from, Coordinate to)** method calculates the angle between two coordinates and rounds it to the nearest multiple of 10 and returns it as an int.

- Public static **calculateNewPos(Coordinate from, double distance, int angle)** calculates the new position you obtain when moving with the provided angle the given distance and returns it as a Coordinate.
- Public static **adjustAngle(int initialAngle, int otherAngle, boolean subtract)** adjusts the initial angle by the other angle, either adding or subtracting based on subtract and returns the new angle as an int.

2.3 BackEndService.java

The BackendService combines some methods related to url routing and reading responses. It's used as a base class for the other services and contains the following attributes:

- A protected final **baseUrl** of type URL, which is the base url passed into the constructor. It is used to go back to a common starting point when routing between different url addresses.
- A protected **url** of type URL, which is the current url in use.
- A protected **connection** of type HttpURLConnection, which is used to connect to the web server.

The following methods are provided:

- Protected **setupNewUrl(String url)**, which navigates to a new url given the url.
- Protected **readResponse()** which reads the response from the current url and returns it as a string.

2.4 ObstacleService.java

The ObstacleService is responsible for reading the obstacles from the webserver. It extends the BackeEndService and has the following additional attribute:

- A private List of LinearRings, **obstacles**, which contains the obstacles.

The following methods are provided:

- Private **addObstacles()**, which returns a List of LinearRings. It reads the response from the webserver, converts them into a FeatureCollection, and invokes **toLinearRing**, to convert each feature into a LinearRing and adds it to the list.
- Private **toLinearRing(Feature feature)**, converts a GeoJSON Feature into a JTS LinearRing, and returns the resulting LinearRing.
- Public **getObstacles()**, which simply returns the obstacles attribute.

2.5 SensorService.java

The SensorService is responsible for reading the sensors for a given day and returning them as a List of Sensor objects. It extends the BackEndService and has the following additional attributes:

- A private static final double **MIN_BATTERY**, which contains the minimum battery capacity needed for a sensor reading to be valid.
- A private final Gson **gson**, which is a Gson object used to read Json.
- A private HashMap<String, Sensor> **sensors** which is used to store the mapping from a what3words address to a sensor.

The following methods are provided:

- Private **retrieveAllSensors()**, which retrieves all sensors from the current url and returns them as a HashMap, where the what3words address is the key and the sensor the value. It converts the read response into a JsonArray using the gson attribute, and then calls **jsonToSensor** to convert each element into a sensor.

- Private **jsonToSensor(JsonElement rawSensor)** converts a JsonElement into a Sensor object and returns it. It does so by reading the attributes of the JsonObject of the rawSensor and then initializing the Sensor object with those values. To get the position of the sensor, which is not provided by the rawSensor, we invoke **getCoordinate** with the what3words location of the sensor. Furthermore, it also invalidates the reading if the battery attribute is less than MIN_BATTERY.
- Private **getCoordinate(String sensorLocation)** returns a Coordinate given a what3words location. It does so by splitting up the address into its parts, navigating to the corresponding url and reading the response.
- Public **getSensors()** returns a List of Sensors stored in **sensors.**

2.6 DroneLogger.java

The DroneLogger is an abstract class, which defines a common interface for all other loggers and contains the following attributes:

- A protected Coordinate, **position**, which is the position the logger received last.
- A protected FileWriter, **file**, which is used to log. It is initialized in the constructor with the corresponding logging path.

It furthermore defines the following abstract methods:

- Public **log(Coordinate newPos, Sensor read_sensor)**, which logs the needed information.
- Public **close()**, which closes the file and cleans up everything needed.

2.7 FlightPathLogger.java

The FlightPathLogger is responsible for logging the flight path of the drone. It extends the DroneLogger and defines the following additional attribute:

- Private int **lineNbr**, the line number written to the output file.

The following methods are provided:

- Public **log(Coordinate newPos, Sensor read_sensor)**, which logs the line number, previous position, new position, direction the drone took as well as the location of the read sensor. As the direction is not provided, it calculates it using the **calculateAngle** method from the angle class. It formats those variables by calling **format** and writes it using the file attribute.
- Private **format(Coordinate newPos, int direction, String location).** It returns a String formatted according to the coursework specifications for the flight path.

```
1,-3.1878,55.9444,110,-3.187902606042998,55.94468190778624,hurt.green.filer
```

lineNbr, prev longitude, prev latitude, direction, new longitude, new latitude, location of read sensor

- Public **close()** invokes the close method on the file attribute.

2.8 ReadingLogger.java

The ReadingLogger is responsible for logging the GeoJSON representation, which includes the read sensors as well as the flight path as a line. It extends the DroneLogger and defines the following additional attributes:

- A private List of Coordinates, **flightPath**, which stores the flight path.
- A private HashMap<String, Feature>, **markers**, which stores all the markers as value of the what3words location.

The following methods are provided:

- Public **log(Coordinate newPos, Sensor read_sensor).** It adds the **newPos** to the **flightPath**, and if the read_sensor is not null, invokes **updateMarkerProps** with the **read_sensor** as argument.
- Private **updateMarkerProps(Sensor read_sensor)**. It gets the marker feature from the **markers** attribute by providing the what3words location of the sensor. Then, it updates the marker feature with the values in the read_sensor object.
- Public **close()** converts the **flightPath** into a List of GeoJSON Points, creates a LineString out of them and stores it together with the **markers** in a FeatureCollection. Then, it writes this FeatureCollection using **file**, and finally closes the **file**.

2.9 Drone.java

The Drone class is responsible for visiting all sensors. It has the following attributes:

- Private static final int **MAX_MOVES**, which defines the maximum number of moves allowed.
- Private static final double **SENSOR_RADIUS**, which defines the maximum distance from which a sensor can be read.
- Private static final double **MOVE_LENGTH**, which defines the length travelled in one move.
- Private Coordinate **startPosition,** which is the start position of the drone.
- Private RoutePlanner **routePlanner.** This planner tells us where to go next given a waypoint coordinate
- Private Map **map,** the map the drone is using. It is used to verify moves the drone makes.
- A private List of DroneLoggers, **loggers**, which stores all loggers that need to be called after each move.
- A private HashSet of Coordinates, **visited**, which stores the visited coordinates. It's used to not go to the same coordinate twice
- Private int **numMoves**, which stores the number of moves taken.

It provides the following methods:

- Public **visitSensors().** This method is responsible for visiting all sensors. It does so by invoking the **getNextPath** method of the **routePlanner** to get the path from a reference coordinate to the next waypoint, and then uses the **navigate** method to move from one coordinate in the route to the next until it reaches the next waypoint. As soon as it reaches one waypoint, it calls **getNextPath** again until the position is close to the original position of the drone. It always keeps track of a reference coordinate, which are the coordinates returned by the **routePlanner**, as well as a current coordinate. We need the reference coordinate to query for the next route, as the current coordinate will only be close to the reference coordinate, but not necessarily the same.
- Private **navigate(Coordinate from, Coordinate to)** is responsible to navigate from one coordinate to another. It does so by repeatedly calling **getNextValidCoord** until we are close to the **to** coordinate. Furthermore, we add the coordinate we moved to the **visited** HashSet. It returns the final Coordinate.
- Private **getNextValidCoord(Coordinate from, Coordinate to)** returns a coordinate which is inside the navigation area, is not in the **visited** set, and, if possible, gets us closer to the **to** coordinate. It does so by trying to go there in a straight line, and if this is an invalid move, it tries to go in an angle 10 degrees smaller, and then 10 degrees larger, oscillating back and forth with a slightly larger angle difference, until we find a valid move.
- Private **log(Coordinate position, Coordinate targetPos)** is used to parse the necessary information to the loggers. Therefore, it checks if the **position** is close to the **targetPos**, and if the **targetPos** is an instance of type Sensor, it collects this sensor as reading. This works as the Coordinates returned by the route planner contain down-casted Sensors.

2.10 Map.java

The Map class is a representation of the valid navigation area. It inherits from the JTS Polygon class and defines the following attribute:

- Private static GeometryFactory **geomFact**, is the geometry factory used to create the Map object.

It contains the following methods:

- Public **verifyMove(Coordinate from, Coordinate to**), returns a boolean which is true if the line spanned by the from and to coordinate is inside the navigation area.
- Private static **createShell()** returns the LinearRing of the outer boundary from the boundary coordinates given in the coursework specification.
- Private static **getHoles(List<LinearRing> obstacles)** filters the provided obstacles and returns an array of LinearRings which are completely covered by the polygon that can be created using the LinearRing of **createShell**.
- Private **alignShell(LinearRing obstacle)** integrates an obstacle into the outer shell. It does so by computing the intersection lines between the obstacle and the shell, ordering them so they form one ongoing line by calling **orderLines()**, and then inserts the line after the closest coordinate to the start of the line.
- Private **getClosestCoordinate(List<Coordinate> coordinates, Coordinate coordinate)** finds the closest coordinate in **coordinates** to **coordinate** and returns its index.
- Private **orderLines(MultiLineString lines)** orders the lines in the MultiLineString such that the ending coordinate of one line is the closest to the starting coordinate of the next. It returns the ordered lines as a List of LinearRings.

2.11 Sensor.java

The Sensor class represents a sensor in the real world and extends the JTS Coordinate class. It defines the following attributes:

- A private String **location**, which is the what3words address of the sensor.
- A private float **battery**, which represents the battery status of the sensors.
- A private Double **reading**, which is the reading the sensor has.

And it defines the following methods:

- Public **getLocation()**, which returns the location
- Public **getReading()**, which returns the reading

Furthermore, it overrides the **hashCode** and **equals** methods to compare different sensors according to their **location** attribute. Finally, it overrides the **toString** method to return a string containing all the attributes of the sensor.

2.12 RoutePlanner.java

The RoutePlanner is the “brain” of my drone control algorithm. It wires all the required classes together. It defines the following attributes:

- A private HashMap<Coordinate, Integer>, **waypoints,** is used to map a waypoint coordinate to the index it had in the list inserted in the constructor.
- A private int[], **route**, it's a mapping where the I-th element gives us the next waypoint index from the I-th waypoint when we follow the route.
- A private Map **map,** is the map object used for validating moves
- A private VisibilityGraph **visibilityGraph,** is used to calculate the shortest path from one waypoint to another
- A private PathFinder **pathFinder**, which is used to find the shortest path between two waypoints.

It defines the following methods:

- Public **getNextPath(Coordinate waypoint)** gives us the next path in form of a List of coordinates given a waypoint coordinate. Therefore, it queries the **waypoints** HashMap to get the index of the coordinate, and then gets the corresponding path we want to follow by querying **getShortestPath** with our current waypoint index and the next waypoint index. The next waypoint index is obtained by querying **route** with the current waypoint index.
- Private **calculateDistances(List<Coordinate> waypoints)** calculates the shortest distances between all waypoints. It does so by adding all waypoints to the **visibilityGraph**, passing the graph into a **pathFinder** instance which finds the shortest distances and updates the values in the visibility graph, and then querying the distances from the **visibilityGraph**. It returns a 2D array of doubles, where the entry in row index I and column index j contains the cost to reach waypoint j from I.

2.13 VisibilityGraph.java

The **VisibilityGraph** is used to store the distances between the vertices of the navigation area as well as additional vertices which are later inserted. It implements the **Graph** interface and defines the following attributes:

- Private Map **map**, used to check if the edge between two vertices is inside the navigation area
- Private List<Coordinate> **additionalCoordinates** includes all coordinates added to the visibility graph after the constructor
- Private List<List<Double>>, **distances**, the distance matrix

Besides implementing the getters and setters for the **Graph** interface, it defines the following methods:

- Public **getAllCoordinates()** returns a List of Coordinates that includes all coordinates of the map as well as the **additionalCoordinates**
- Public **addCoordinate(Coordinate newCoordinate)** adds a new coordinate to the **additionalCoordinates** and calculates the distances of this coordinate to all other and adds them to the **distanceMatrix**.
- Private **constructVisibilityGraph()** initializes and sets the attributes of the graph.
- Private void **initDistMatrix()** initializes the **distances** matrix so they are 2 dimensional arrays.
- Private void **connectEdges()** computes the edge distances of edges in each of the boundaries in the **map**.

2.14 PathFinder.java

The PathFinder class is responsible for finding the shortest path between all nodes in a **Graph**. It defines the following attributes:

- Private final Graph **graph**, which is the graph used to get and set distances.
- A List<List<List<Coordinate>>>, **paths**, where the List at row index I and column index j holds the path to get from node I to node j.

It defines the following methods:

- Public **getShortestPath(int fromIdx, int toIdx)** which returns the list of coordinates at row index I and column index j.
- Private **calculateShortestPaths()**, which calculates the shortest paths between all node pairs. It employs the Floyd-Warshall algorithm to do so. It updates the distance between two nodes in the **graph**, while adding the shortest paths to **paths**.

2.15 RouteFinder.java

The RouteFinder is used to find a short route visiting all nodes in a graph while returning to the start position. It communicates with the graphhopper library to find that route. It defines the following attributes:

- Private final String **startLocation**, which is the start location of our graph traversal. It is initially set to 0.
- Private double[][] **distanceMatrix,** which is the distance matrix between all nodes
- Private VehicleRoutingAlgorithm **routing**, the instance which we query to get a short route.

It defines the following methods:

- Public **findShortestRoute(),** which finds the shortest route visiting all sensors. It returns an array of indices representing the order in which to visit the nodes.
- Private void **setupRouting()** calls all methods used to setup the **RouteFinder** object.
- Private **constructVehicle()** constructs a VehicleImpl object which is used in the graphhopper library to represent an object visiting all nodes.
- Private **constructCostMatrix()** constructs the cost matrix used in the GraphHopper library.
- Private **constructRouter(VehicleRoutingTransportCostsMatrix costMatrix, VehicleImpl vehicle)** constructs the **routing** object.

2.16 MarkerProperties.java

The MarkerProperties Enum is used to store the marker properties of each marker. Besides getters and setters, it implements the following method:

- Public static **FromAirPollution(double pollution)** which based on the pollution level will return the corresponding marker properties.

3. Drone control algorithm

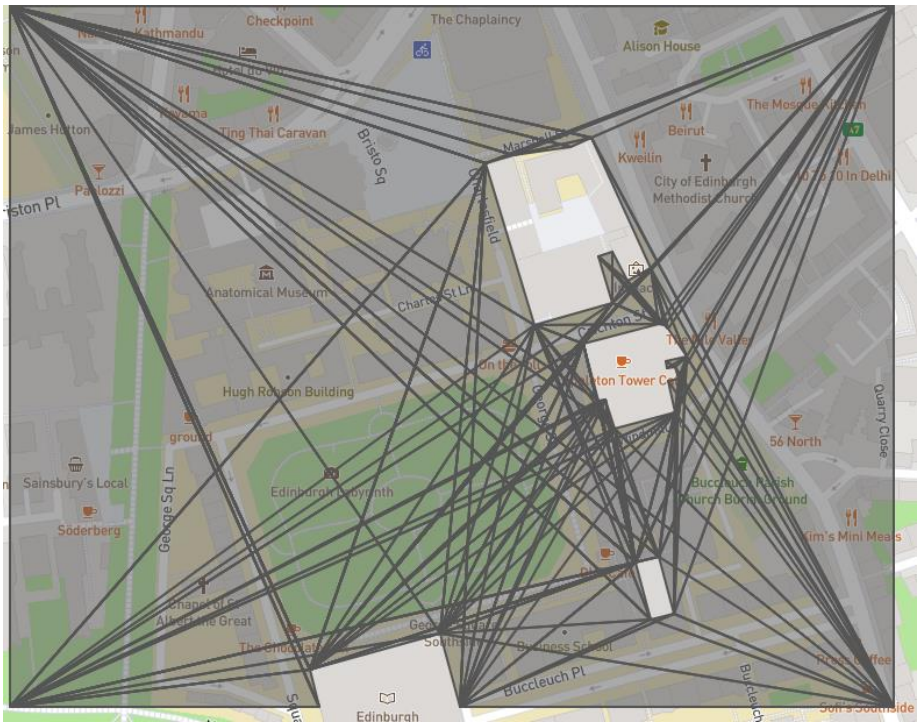
3.1 Short description

In essence, we find a short path in the following way. First, we find the shortest distances and paths between all waypoints. When two waypoints cannot be connected by a straight line as an obstacle is in the way, we find the shortest possible path by going over the edges of the obstacles. Then, we run an optimization algorithm to find an overall short path which visits all waypoints. After this, we have the order in which we must visit the waypoints.

Then to visit the waypoint, we follow the path previously calculated to get from one waypoint to the other, by getting close to each of the coordinates in the path in the order provided. In the case we cannot reach the next coordinate from our current because an obstacle is in the way, we oscillate between slightly larger angles to the left and right of the initial angle between the two waypoints until we can make a valid move.

3.2 Finding paths between waypoints when an obstacle is in the way

As described earlier, to get from one waypoint to the other in the shortest possible way, we need to go over the edges of our obstacles if we cannot reach the other waypoint in a straight line. Therefore, I created a visibility graph of the obstacle vertices, where there is an edge between two vertices if they can be connected by a straight line while not going out of the navigation area. We then inject all waypoints as additional waypoints into our **VisibilityGraph** and create additional edges to every vertex than can be reached directly from them. Then, we feed the **VisibilityGraph** into our **PathFinder**, which finds the shortest paths between all vertices using the Floyd-Warshall algorithm.



Underlying visibility graph

3.3 Shortest path visiting all sensors

Assuming we constructed a distance matrix where the I-th row and j-th column contains the shortest distance to get from the I-th waypoint to the j-th, we can construct a short path by using an algorithm that performs well on the traveling salesperson problem. For this, I am using the GraphHopper Jsprit library, which already defines many useful classes for dealing with such problems. The **RouteFinder,** when given a distance matrix, calculates the shortest route which visits all nodes in the distance matrix. The algorithm used involves the *ruin-and-recreate* principle, which is a large neighborhood search that combines elements of simulated annealing and threshold-accepting algorithms (Schrimpf et al. [2000, pg. 142]) For more information about the algorithm, please visit the GraphHopper Jsprit GitHub page.

3.4 Handling positional approximations

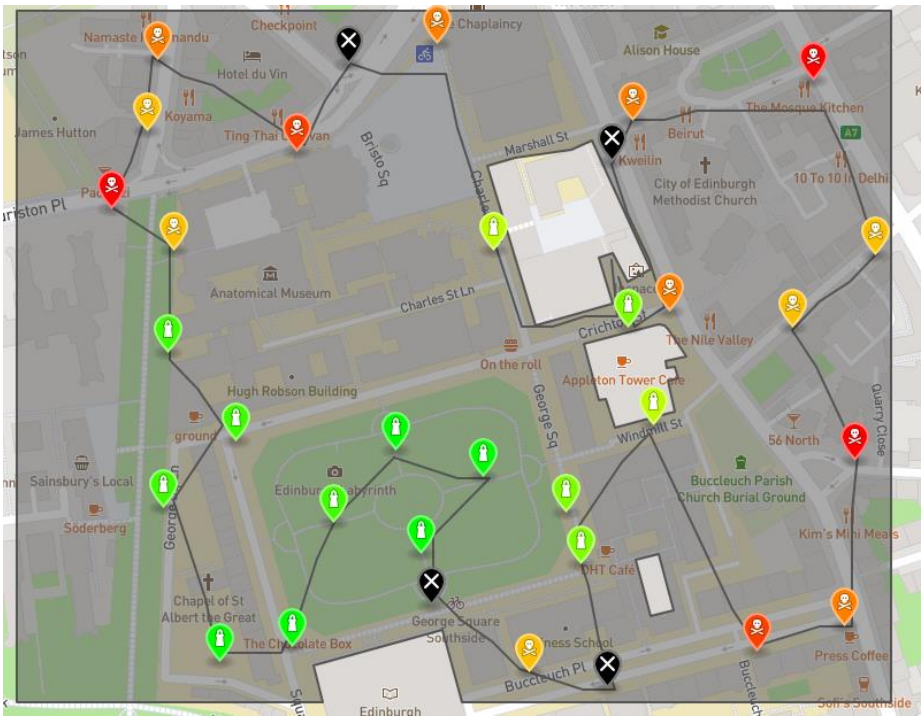
In theory, if we directly follow each coordinate in the provided route, we will never leave the navigation area and only touch the boundaries of the obstacles. As we are only guaranteed to be in the vicinity of the provided coordinates, this might not be the case. Therefore, we need some way of choosing a different direction, which is not necessarily the angle between the current coordinate and the coordinate we want to get to. I do so by oscillating between angles which are in the positive and negative direction relative to the initial angle, as they get us closest to the target coordinate. The issue with that is, that sometimes we jump back and forth between different positions, as illustrated in the example below.

(12-12-2021 without modification)

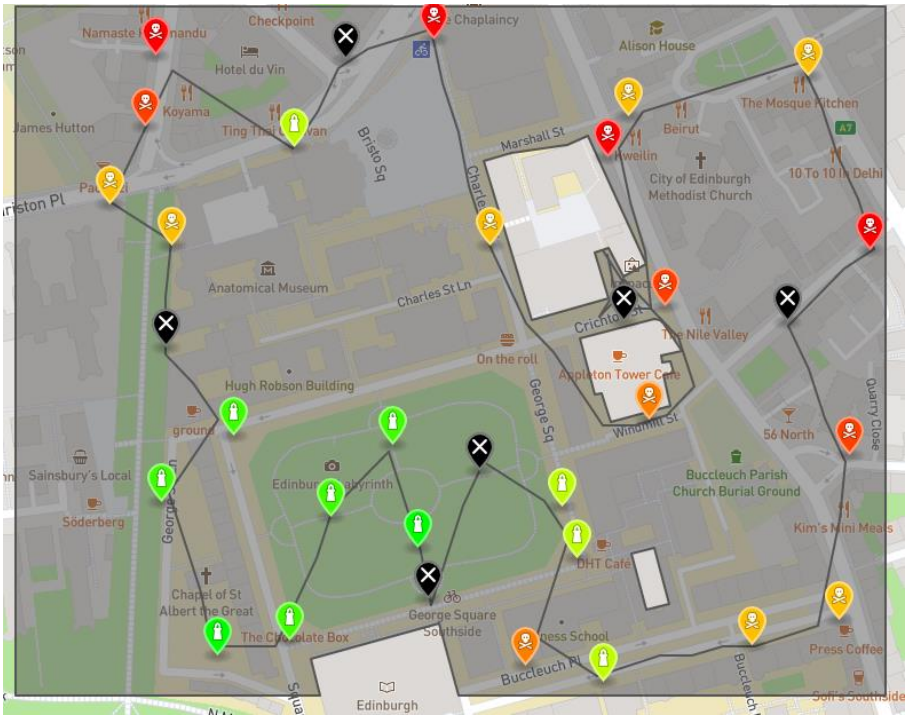
To circumvent this, I use a HashSet of coordinates that stores all previously visited locations and ignore those when trying to get closer to a target coordinate.

3.5 Results

The following results were obtained when initializing the drone at a longitude and longitude of -3.186924308538437 and 55.9449287211836. I chose this location as it is directly between obstacles which makes it difficult to navigate.



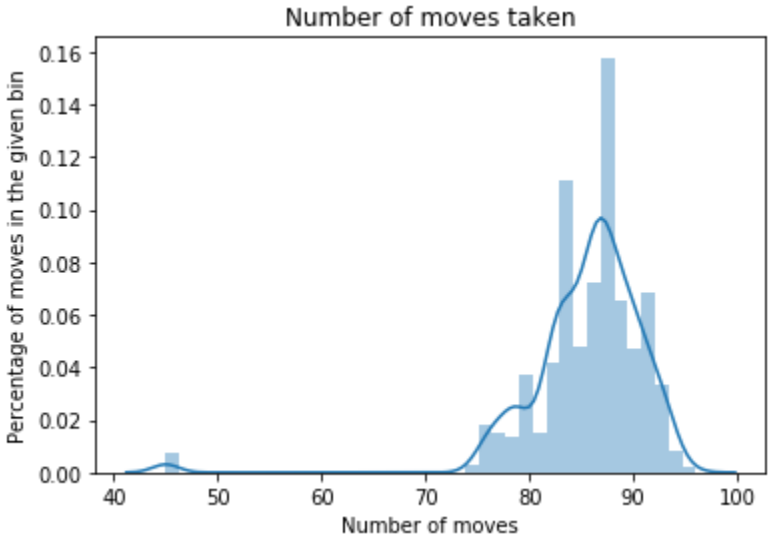
(12-12-2021)



(08-09-2020)

The drone generated an efficient flight path and when it was not able to fly close to the starting location as it would hit the boundary, it tried again from another position. Also notice how close the drone maneuvers along Appleton tower, saving many moves in the process.

I tested my drone on most of the provided files on the webserver. It managed to visit all the waypoints and returned safely to its starting position and took on average 85.5 moves to do so. Looking at the histogram, it does so consistently.



4. References

- Mapbox SDK
- JTS Geometry library
- Jsprit routing library
- Floyd-Warshall algorithm