ILP - Coursework

Autonomous Drone Application that Collects Air Quality Data from Sensors

s1870697

December 2020

Contents

1 Software Architecture

The application has classes each with their own methods and attributes to run the program combined altogether, whilst abstracting the functionalities when other objects want to use it. These interactions between different instances of the classes are made possible through some of the concepts of Object-Oriented design. Figure 1 shows the relationships between the different classes in the program.

In terms of the structure itself, the architecture is divided into separate buckets, each representing a hierarchy of the goal of the classes in those buckets. See Figure 2, where classes in Drone module are responsible for elements pertaining to our simulated autonomous drone. The report goes in depth about the choices made behind choosing these classes for the software and the communication models.

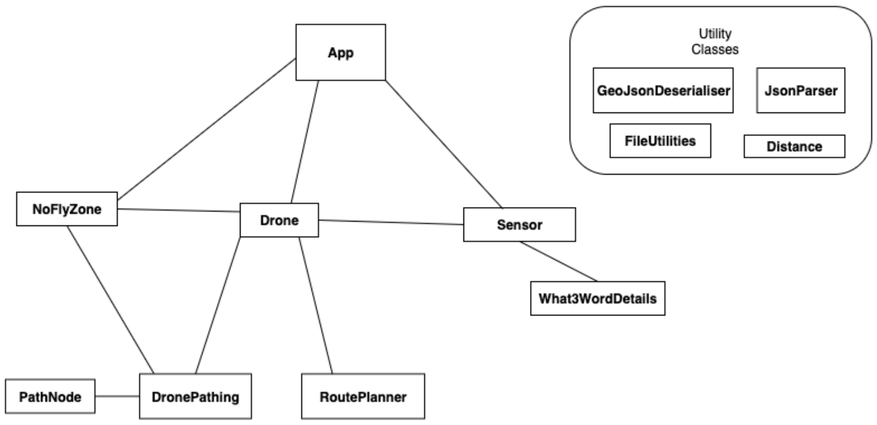


Figure 1: Software Architecture - Class Relationships

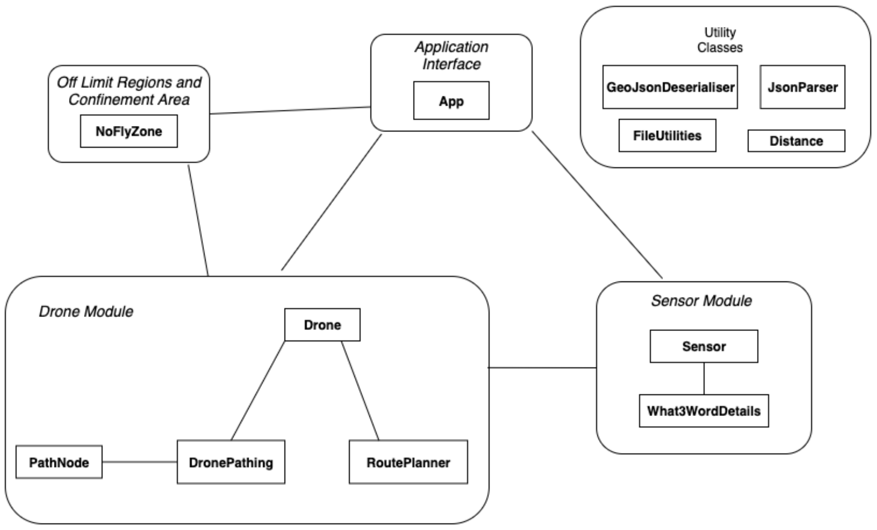


Figure 2: Software Architecture - The structure of the drone program

The bulk of the software lies mainly at the central part which is the Drone module after the Drone object retrieves information on the sensor from webserver through App, with the JsonParser static class handling that process. To go through the process of how the software works overall, it is best illustrated with this overview sequence diagram in Figure 3 from the start of the software towards generating new paths for the drone to take. Note that the sequence is an oversimplification of the entire program, but it represents the bulk of the operations that the crucial objects in this software handles. It starts off with receiving arguments on the starting point of the drone (while making sure it is a valid point), the date, the randomSeed, and the correct portNumber inputted to allow us to connect to the webserver successfully. This is the main functionality of the App class. The App class also briefly parses some of the information needed to pass it to the later on newly created instances of objects that use them. (As a side note, my implementation does not make use of the random seed since it was not needed as the algorithm reproduces for the same arguments provided.)

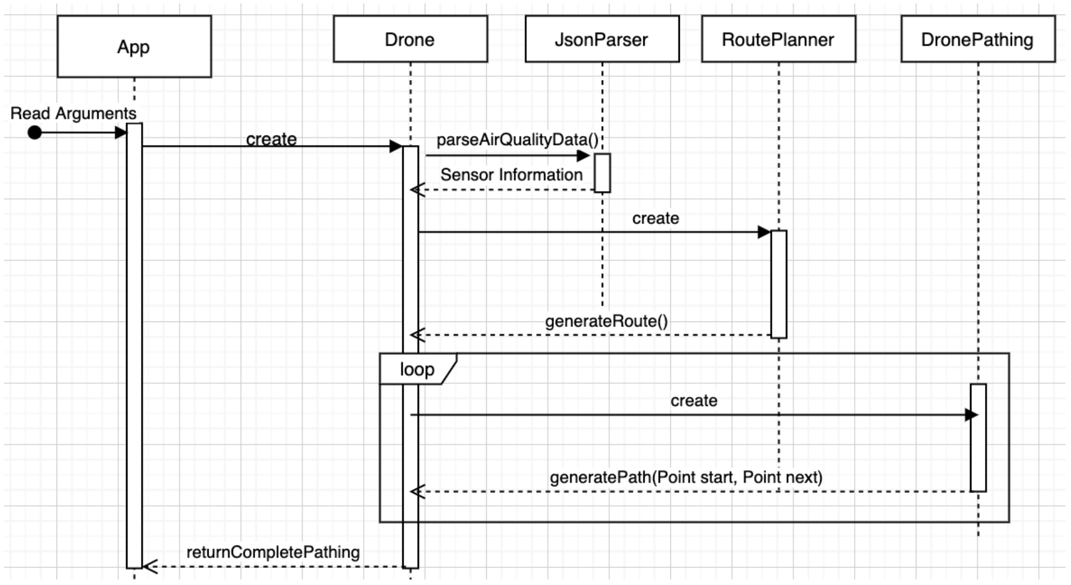


Figure 3: *Sequence Diagram - Abstract Flow of Program for Generating Routes and complete pathing for Drone to take.*
Note that generateRoute() and generatePathAStar() from one point to another here represent two different things. Route is involved with finding the route for sensors for the entire trip ignoring areas and buildings we are meant to avoid. On the other hand, pathing is involved with finding the best path from one point to another while avoiding obstacles previously ignored (the off limit areas). This is talked about in depth in *Section 3 - Drone Control Algorithm* of this report, and the reason for this selection of classes RoutePlanner and DronePathing are for the purposes of finding the best pathing for the drone, each handling a different problem.

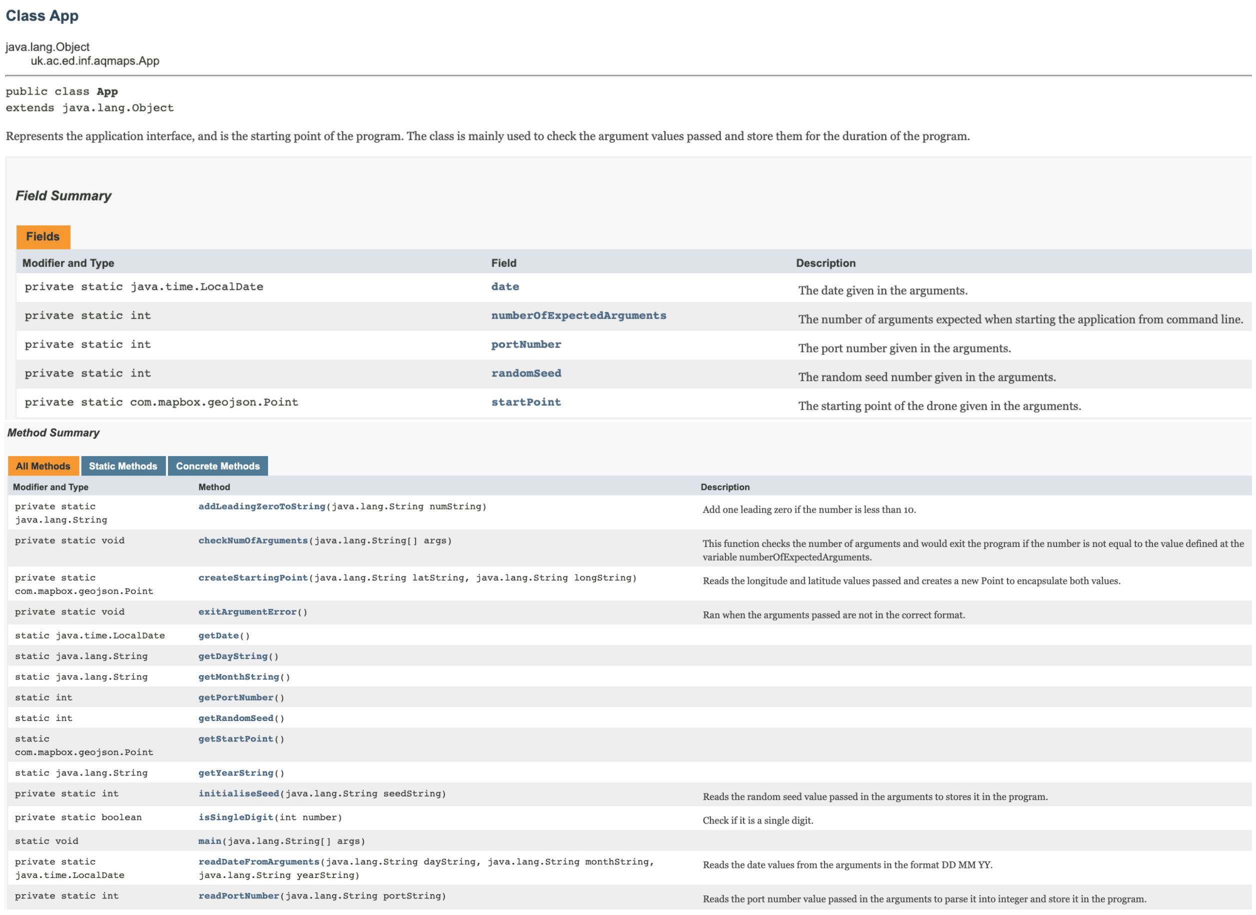
The **Drone** object created contains attributes to define its movement and other properties along with other external information such as the sensors and the off limit areas it needs to stay away from. For generating the route for the drone to take, the **Drone** object requests the journey details by communicating with instances of **DronePathing** and **RoutePlanner** classes as a client-server model, with the server being the Drone. It uses the outputs returned by them afterwards to read sensor data when it is in range at the end of a part of the entire flight journey. The reason why two classes are made for handling the predicted flight journeys is because they handle different problems (See Figure 2 Caption and Section 3 of the report). In addition, the reason for separating these algorithms instead of having them in the **Drone** class is to enable readability and maintenance of code in the future. If the algorithms prove to be not efficient and need to be optimised if the scenario of the task needed by the researcher changes, the main algorithms for detecting best flight paths are detached from **Drone** class. The **Drone** object is mainly concerned about reading sensors and emulating the path generated by the algorithms by going from one point to another rather than finding that route. Finally, the **PathNode** class in our drone module is used to help with the pathfinding A* search algorithm that was implemented, by storing information which helps us keep track of the pathing generated by the algorithm.

The sensor module is involved with everything related to sensors, including communicating with the webserver through JsonParser to get the coordinates given its what3word details.

The utility classes are a set of classes that contain related methods, to be reused across the software if necessary.

2 Class Documentation

2.1 App



Class App

java.lang.Object
    uk.ac.ed.inf.aqmaps.App

```
public class App
extends java.lang.Object
```

Represents the application interface, and is the starting point of the program. The class is mainly used to check the argument values passed and store them for the duration of the program.

Field Summary

Fields

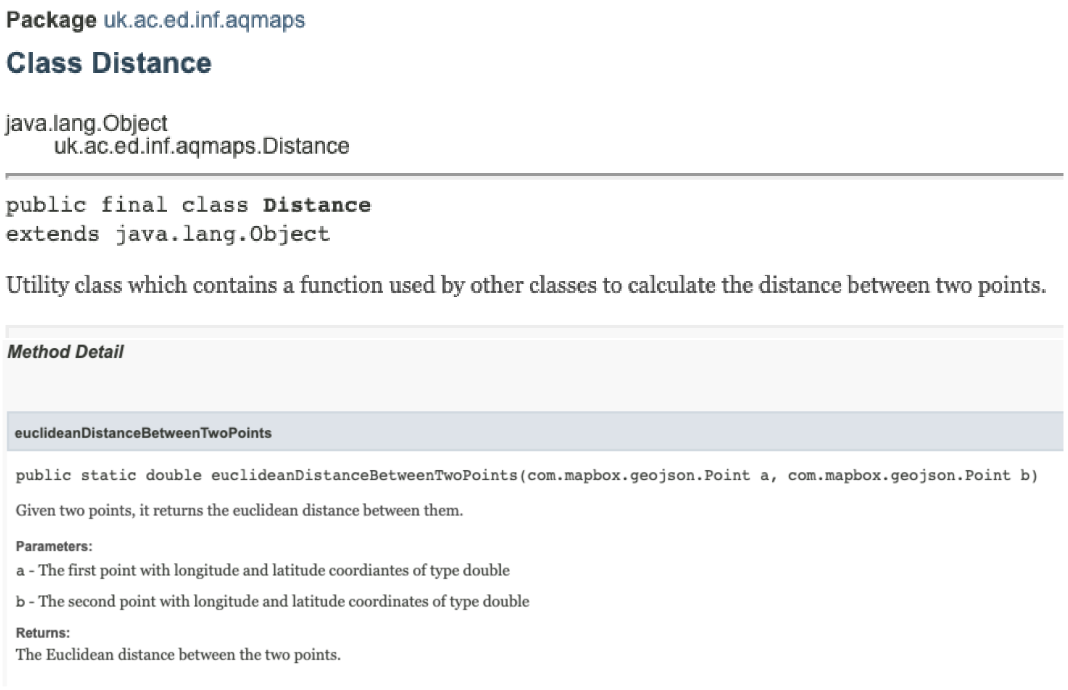
Modifier and Type	Field	Description
`private static java.time.LocalDate`	`date`	The date given in the arguments.
`private static int`	`numberOfExpectedArguments`	The number of arguments expected when starting the application from command line.
`private static int`	`portNumber`	The port number given in the arguments.
`private static int`	`randomSeed`	The random seed number given in the arguments.
`private static com.mapbox.geojson.Point`	`startPoint`	The starting point of the drone given in the arguments.

Method Summary

All Methods | Static Methods | Concrete Methods

Modifier and Type	Method	Description
`private static java.lang.String`	`addLeadingZeroToString(java.lang.String numString)`	Add one leading zero if the number is less than 10.
`private static void`	`checkNumOfArguments(java.lang.String[] args)`	This function checks the number of arguments and would exit the program if the number is not equal to the value defined at the variable numberOfExpectedArguments.
`private static com.mapbox.geojson.Point`	`createStartingPoint(java.lang.String latString, java.lang.String longString)`	Reads the longitude and latitude values passed and creates a new Point to encapsulate both values.
`private static void`	`exitArgumentError()`	Ran when the arguments passed are not in the correct format.
`static java.time.LocalDate`	`getDate()`	
`static java.lang.String`	`getDayString()`	
`static java.lang.String`	`getMonthString()`	
`static int`	`getPortNumber()`	
`static int`	`getRandomSeed()`	
`static com.mapbox.geojson.Point`	`getStartPoint()`	
`static java.lang.String`	`getYearString()`	
`private static int`	`initialiseSeed(java.lang.String seedString)`	Reads the random seed value passed in the arguments to stores it in the program.
`private static boolean`	`isSingleDigit(int number)`	Check if it is a single digit.
`static void`	`main(java.lang.String[] args)`	
`private static java.time.LocalDate`	`readDateFromArguments(java.lang.String dayString, java.lang.String monthString, java.lang.String yearString)`	Reads the date values from the arguments in the format DD MM YY.
`private static int`	`readPortNumber(java.lang.String portString)`	Reads the port number value passed in the arguments to parse it into integer and store it in the program.

2.2 Distance



Package uk.ac.ed.inf.aqmaps

Class Distance

java.lang.Object
    uk.ac.ed.inf.aqmaps.Distance

```
public final class Distance
extends java.lang.Object
```

Utility class which contains a function used by other classes to calculate the distance between two points.

Method Detail

euclideanDistanceBetweenTwoPoints

```
public static double euclideanDistanceBetweenTwoPoints(com.mapbox.geojson.Point a, com.mapbox.geojson.Point b)
```

Given two points, it returns the euclidean distance between them.

Parameters:

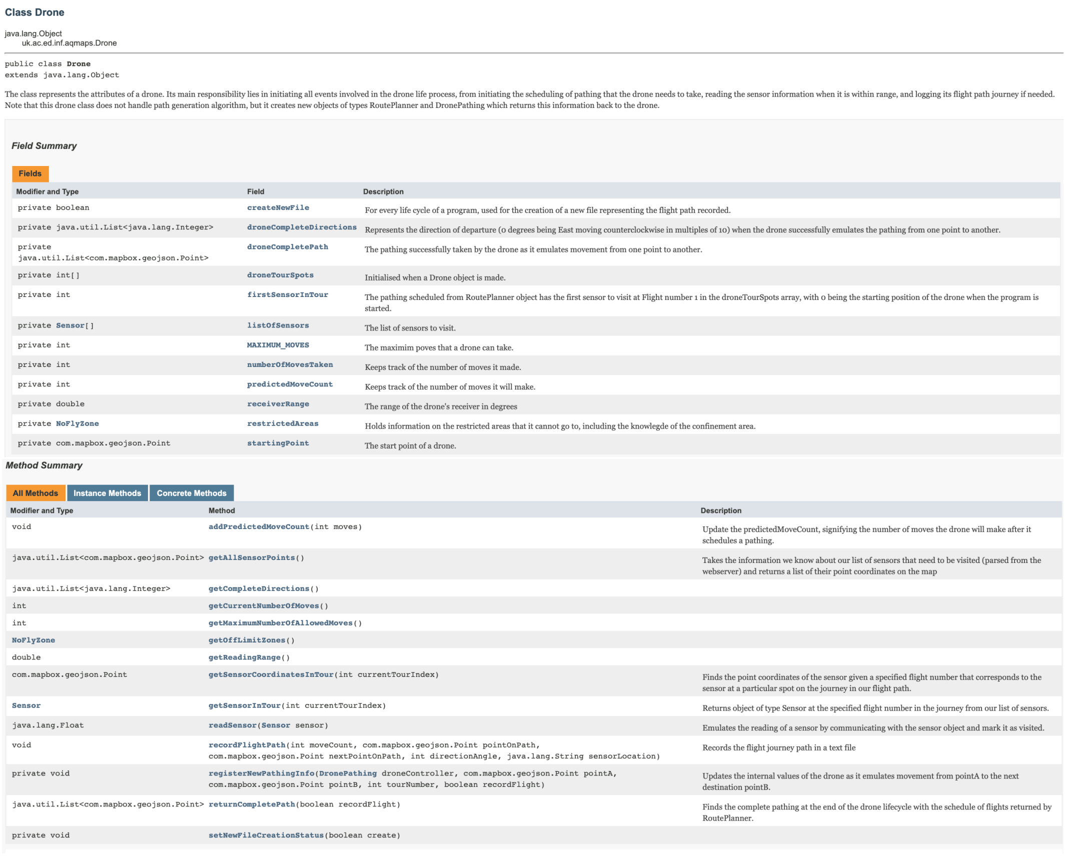
a - The first point with longitude and latitude coordiantes of type double

b - The second point with longitude and latitude coordinates of type double

Returns:

The Euclidean distance between the two points.

2.3 Drone



Class Drone

java.lang.Object
    uk.ac.ed.inf.aqmaps.Drone

```
public class Drone
extends java.lang.Object
```

The class represents the attributes of a drone. Its main responsibility lies in initiating all events involved in the drone life process, from initiating the scheduling of pathing that the drone needs to take, reading the sensor information when it is within range, and logging its flight path journey if needed. Note that this drone class does not handle path generation algorithm, but it creates new objects of types RoutePlanner and DronePathing which returns this information back to the drone.

Field Summary

Fields

Modifier and Type	Field	Description
private boolean	createNewFile	For every life cycle of a program, used for the creation of a new file representing the flight path recorded.
private java.util.List<java.lang.Integer>	droneCompleteDirections	Represents the direction of departure (0 degrees being East moving counterclockwise in multiples of 10) when the drone successfully emulates the pathing from one point to another.
private java.util.List<com.mapbox.geojson.Point>	droneCompletePath	The pathing successfully taken by the drone as it emulates movement from one point to another.
private int[]	droneTourSpots	Initialised when a Drone object is made.
private int	firstSensorInTour	The pathing scheduled from RoutePlanner object has the first sensor to visit at Flight number 1 in the droneTourSpots array, with 0 being the starting position of the drone when the program is started.
private Sensor[]	listOfSensors	The list of sensors to visit.
private int	MAXIMUM_MOVES	The maximim poves that a drone can take.
private int	numberOfMovesTaken	Keeps track of the number of moves it made.
private int	predictedMoveCount	Keeps track of the number of moves it will make.
private double	receiverRange	The range of the drone's receiver in degrees
private NoFlyZone	restrictedAreas	Holds information on the restricted areas that it cannot go to, including the knowlegde of the confinement area.
private com.mapbox.geojson.Point	startingPoint	The start point of a drone.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
void	addPredictedMoveCount(int moves)	Update the predictedMoveCount, signifying the number of moves the drone will make after it schedules a pathing.
java.util.List<com.mapbox.geojson.Point>	getAllSensorPoints()	Takes the information we know about our list of sensors that need to be visited (parsed from the webserver) and returns a list of their point coordinates on the map
java.util.List<java.lang.Integer>	getCompleteDirections()	
int	getCurrentNumberOfMoves()	
int	getMaximumNumberOfAllowedMoves()	
NoFlyZone	getOffLimitZones()	
double	getReadingRange()	
com.mapbox.geojson.Point	getSensorCoordinatesInTour(int currentTourIndex)	Finds the point coordinates of the sensor given a specified flight number that corresponds to the sensor at a particular spot on the journey in our flight path.
Sensor	getSensorInTour(int currentTourIndex)	Returns object of type Sensor at the specified flight number in the journey from our list of sensors.
java.lang.Float	readSensor(Sensor sensor)	Emulates the reading of a sensor by communicating with the sensor object and mark it as visited.
void	recordFlightPath(int moveCount, com.mapbox.geojson.Point pointOnPath, com.mapbox.geojson.Point nextPointOnPath, int directionAngle, java.lang.String sensorLocation)	Records the flight journey path in a text file
private void	registerNewPathingInfo(DronePathing droneController, com.mapbox.geojson.Point pointA, com.mapbox.geojson.Point pointB, int tourNumber, boolean recordFlight)	Updates the internal values of the drone as it emulates movement from pointA to the next destination pointB.
java.util.List<com.mapbox.geojson.Point>	returnCompletePath(boolean recordFlight)	Finds the complete pathing at the end of the drone lifecycle with the schedule of flights returned by RoutePlanner.
private void	setNewFileCreationStatus(boolean create)	

2.4 DronePathing

Package uk.ac.ed.inf.aqmaps

Class DronePathing

java.lang.Object
    uk.ac.ed.inf.aqmaps.DronePathing

```
public class DronePathing
extends java.lang.Object
```

Path Finder problem while avoiding obstacles. The class is responsible for generating a pathing from one point to another. It uses Weighted A* search to navigate to the target destination. Some methods are private in order to abstract inner functionalities of this class, which the Drone class does not need to access to.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
`private void`	`backTrackAndStoreResults(PathNode currNode)`	At the end of the A star iteration when close to the target point, back track the journey taken till the starting point and store the direction angles and paths.
`private boolean`	`checkNumberOfMoves(PathNode n)`	Check to see if the number of moves of our drone has reached the limit of moves allowed.
`private java.util.List<PathNode>`	`findNextPossibleNodes(PathNode currNode)`	Given a node, find the next possible nodes, which will be the neighbours.
`private boolean`	`firstMove(PathNode node)`	Edge case: If number of moves zero, we cant take 0 path.
`void`	`generatePathAStar(com.mapbox.geojson.Point startCoords, com.mapbox.geojson.Point targetCoords)`	Weighted A* Search Implementation.
`java.util.List<java.lang.Integer>`	`getDirectionAnglesOfCurrentPathing()`	
`java.util.List<com.mapbox.geojson.Point>`	`getPathOfCurrentMovement()`	
`private double`	`heuristicFunction(com.mapbox.geojson.Point pointA, com.mapbox.geojson.Point endPoint)`	Returns the estimated cost of the heuristic function, which is the euclidean distance from one point to the final target point at the end of the journey.
`private void`	`initaliseNewPathing()`	
`private void`	`initiatePossibleMoveSets()`	Represents the possible directions and the change in displacements that our drone can move to.
`private boolean`	`searchNeighborInList(java.util.List<PathNode> arr, PathNode node)`	Find if the current node is in our list by checking the coordinates.
`private boolean`	`withinTargetRange(com.mapbox.geojson.Point curr, com.mapbox.geojson.Point target)`	Given two points, determine whether the drone will be close to the target at a point in space curr.

heuristicFunction

```
private double heuristicFunction(com.mapbox.geojson.Point pointA, com.mapbox.geojson.Point endPoint)
```

Returns the estimated cost of the heuristic function, which is the euclidean distance from one point to the final target point at the end of the journey. The distance is multiplied by an epsilon cost to speed up the search algorithm and prioritise the traversal of points more closer to the goal.

Parameters:

`pointA` - A point on space

`endPoint` - The final end target point

Returns:

the euclidean distance between the two points

generatePathAStar

```
public void generatePathAStar(com.mapbox.geojson.Point startCoords, com.mapbox.geojson.Point targetCoords)
```

Weighted A* Search Implementation. Finds the shortest path from the current position of the drone to the next position of the next target point. The algorithm also makes sure that a move is made before taking a reading, even if the drone is besides to the target coordinates.

Parameters:

`startCoords` - The start point of the journey

`targetCoords` - The end goal that the algorithm tries to find a path to reach for.

searchNeighborInList

```
private boolean searchNeighborInList(java.util.List<PathNode> arr, PathNode node)
```

Find if the current node is in our list by checking the coordinates. Used in order to avoid adding duplicates to our open or closed list.

Parameters:

`arr` - List of nodes saved in open or closed list to evaluate the variable node with to see if node already exists in one of those lists.

`node` - The current node

Returns:

True if the node coordinates match up with one of the coordinates in our arr (which is either the open or closed list).

searchNeighborInList

```
private boolean searchNeighborInList(java.util.List<PathNode> arr, PathNode node)
```

Find if the current node is in our list by checking the coordinates. Used in order to avoid adding duplicates to our open or closed list.

Parameters:

`arr` - List of nodes saved in open or closed list to evaluate the variable node with to see if node already exists in one of those lists.

`node` - The current node

Returns:

True if the node coordinates match up with one of the coordinates in our arr (which is either the open or closed list).

initiatePossibleMoveSets

```
private void initiatePossibleMoveSets()
```

Represents the possible directions and the change in displacements that our drone can move to. Rotate for a total of 36 including 0 degrees initial state. The values will be used to add to the coordinates of the points.

findNextPossibleNodes

```
private java.util.List<PathNode> findNextPossibleNodes(PathNode currNode)
```

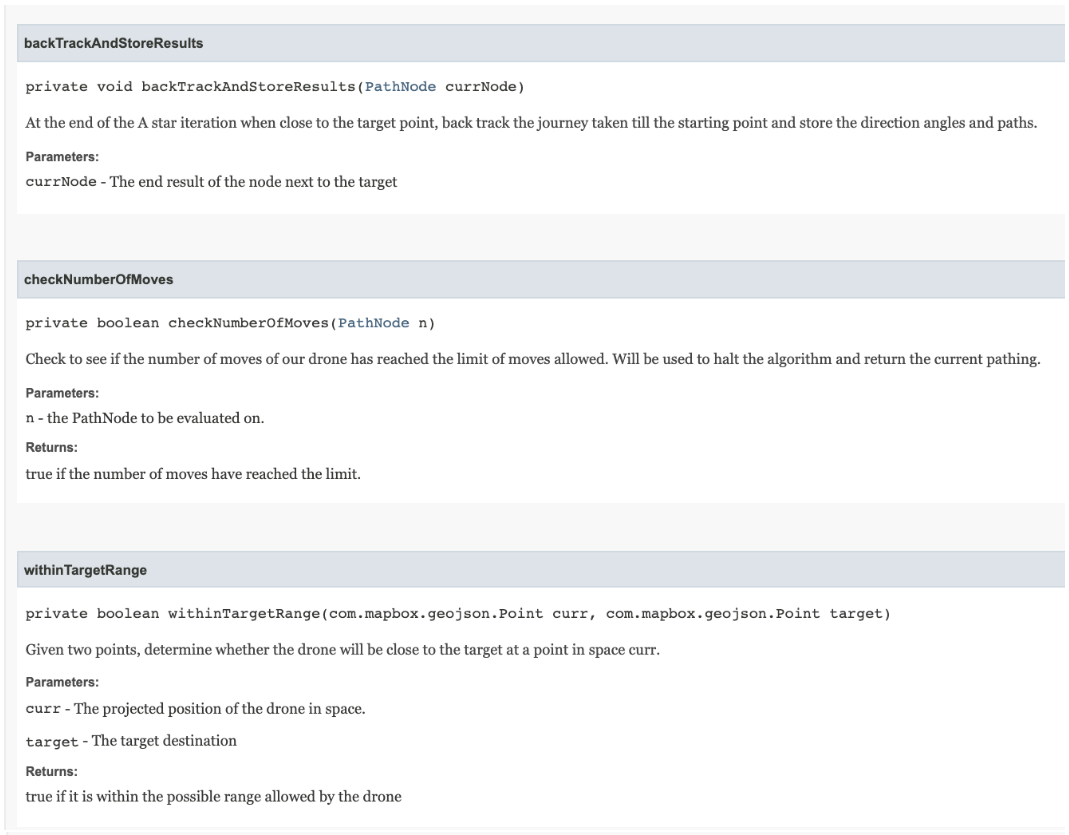
Given a node, find the next possible nodes, which will be the neighbours. The method also checks by not generating new nodes that are outside the confinement area or inside a no fly zone.

Parameters:

`currNode` - The current node that will be used to find next nodes

Returns:

the list of neighbours



backTrackAndStoreResults

```
private void backTrackAndStoreResults(PathNode currNode)
```

At the end of the A star iteration when close to the target point, back track the journey taken till the starting point and store the direction angles and paths.

Parameters:

`currNode` - The end result of the node next to the target

checkNumberOfMoves

```
private boolean checkNumberOfMoves(PathNode n)
```

Check to see if the number of moves of our drone has reached the limit of moves allowed. Will be used to halt the algorithm and return the current pathing.

Parameters:

`n` - the PathNode to be evaluated on.

Returns:

true if the number of moves have reached the limit.

withinTargetRange

```
private boolean withinTargetRange(com.mapbox.geojson.Point curr, com.mapbox.geojson.Point target)
```

Given two points, determine whether the drone will be close to the target at a point in space curr.

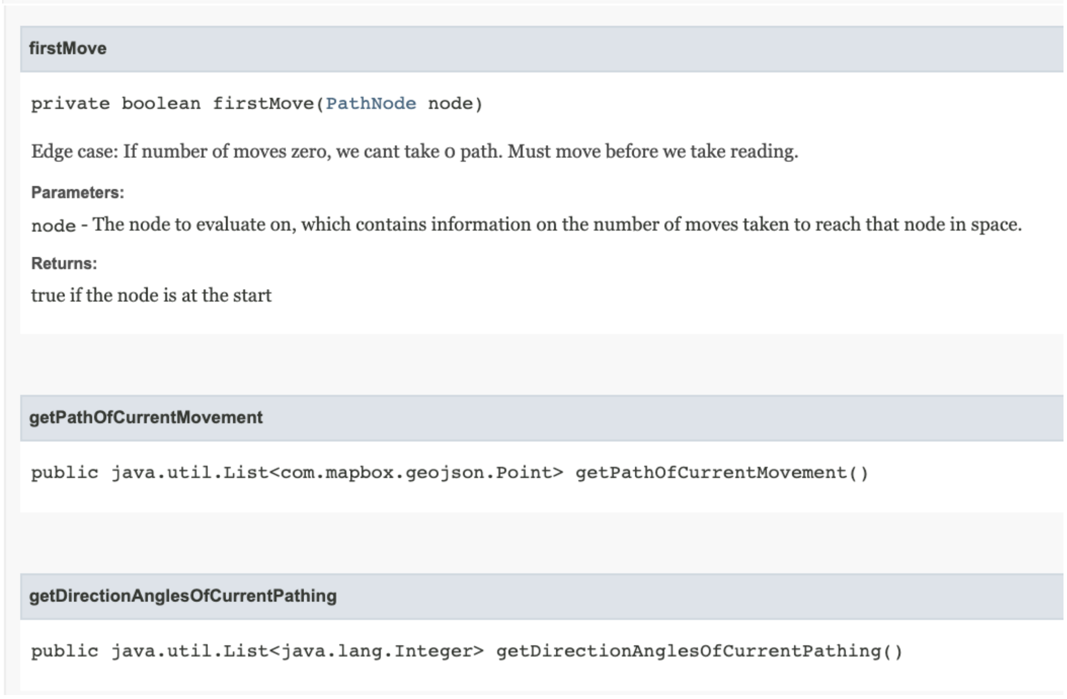
Parameters:

`curr` - The projected position of the drone in space.

`target` - The target destination

Returns:

true if it is within the possible range allowed by the drone



firstMove

```
private boolean firstMove(PathNode node)
```

Edge case: If number of moves zero, we cant take 0 path. Must move before we take reading.

Parameters:

`node` - The node to evaluate on, which contains information on the number of moves taken to reach that node in space.

Returns:

true if the node is at the start

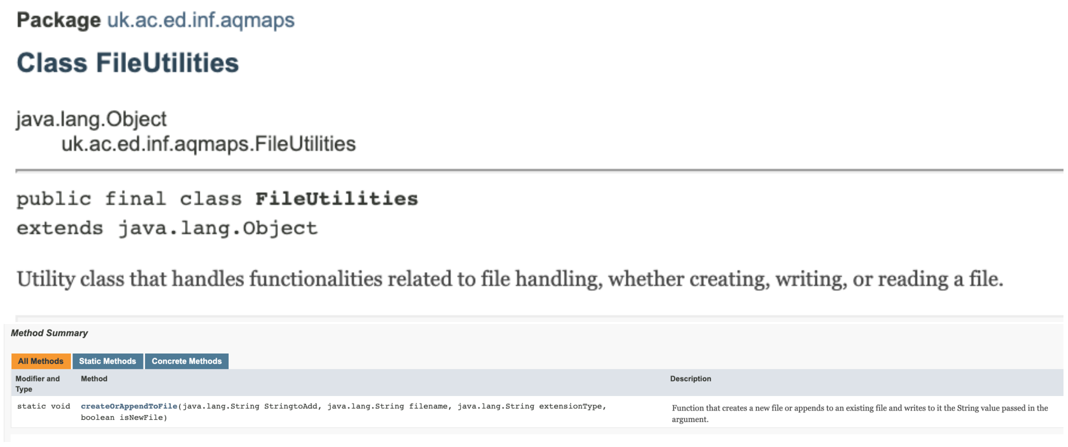
getPathOfCurrentMovement

```
public java.util.List<com.mapbox.geojson.Point> getPathOfCurrentMovement()
```

getDirectionAnglesOfCurrentPathing

```
public java.util.List<java.lang.Integer> getDirectionAnglesOfCurrentPathing()
```

2.5 FileUtilities



Package uk.ac.ed.inf.aqmaps

Class FileUtilities

java.lang.Object
    uk.ac.ed.inf.aqmaps.FileUtilities

```
public final class FileUtilities
extends java.lang.Object
```

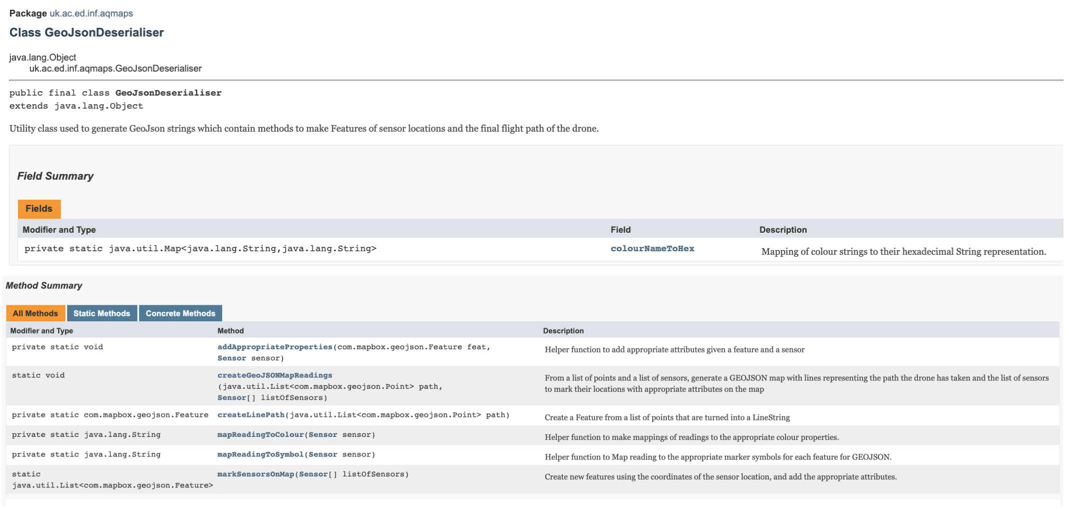
Utility class that handles functionalities related to file handling, whether creating, writing, or reading a file.

Method Summary

All Methods | Static Methods | Concrete Methods

Modifier and Type	Method	Description
`static void`	`createOrAppendToFile(java.lang.String StringtoAdd, java.lang.String filename, java.lang.String extensionType, boolean isNewFile)`	Function that creates a new file or appends to an existing file and writes to it the String value passed in the argument.

2.6 GeoJsonDeserialiser



Package uk.ac.ed.inf.aqmaps

Class GeoJsonDeserialiser

java.lang.Object
    uk.ac.ed.inf.aqmaps.GeoJsonDeserialiser

```
public final class GeoJsonDeserialiser
extends java.lang.Object
```

Utility class used to generate GeoJson strings which contain methods to make Features of sensor locations and the final flight path of the drone.

Field Summary

Fields

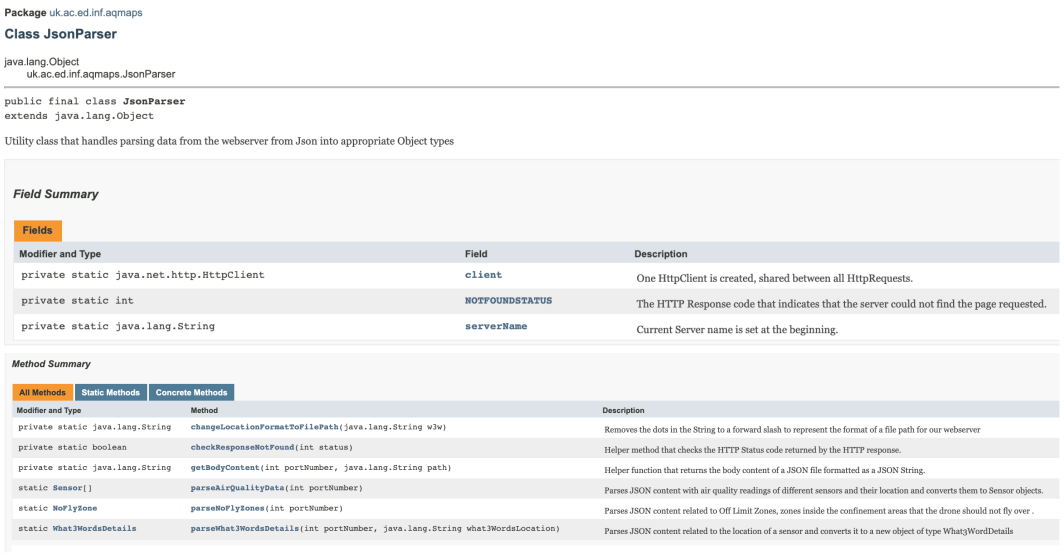
Modifier and Type	Field	Description
`private static java.util.Map<java.lang.String,java.lang.String>`	`colourNameToHex`	Mapping of colour strings to their hexadecimal String representation.

Method Summary

All Methods | Static Methods | Concrete Methods

Modifier and Type	Method	Description
`private static void`	`addAppropriateProperties(com.mapbox.geojson.Feature feat, Sensor sensor)`	Helper function to add appropriate attributes given a feature and a sensor
`static void`	`createGeoJSONMapReadings(java.util.List<com.mapbox.geojson.Point> path, Sensor[] listOfSensors)`	From a list of points and a list of sensors, generate a GEOJSON map with lines representing the path the drone has taken and the list of sensors to mark their locations with appropriate attributes on the map
`private static com.mapbox.geojson.Feature`	`createLinePath(java.util.List<com.mapbox.geojson.Point> path)`	Create a Feature from a list of points that are turned into a LineString
`private static java.lang.String`	`mapReadingToColour(Sensor sensor)`	Helper function to make mappings of readings to the appropriate colour properties.
`private static java.lang.String`	`mapReadingToSymbol(Sensor sensor)`	Helper function to Map reading to the appropriate marker symbols for each feature for GEOJSON.
`static java.util.List<com.mapbox.geojson.Feature>`	`markSensorsOnMap(Sensor[] listOfSensors)`	Create new features using the coordinates of the sensor location, and add the appropriate attributes.

2.7 JsonParser



Package uk.ac.ed.inf.aqmaps

Class JsonParser

java.lang.Object
    uk.ac.ed.inf.aqmaps.JsonParser

```
public final class JsonParser
extends java.lang.Object
```

Utility class that handles parsing data from the webserver from Json into appropriate Object types

Field Summary

Fields

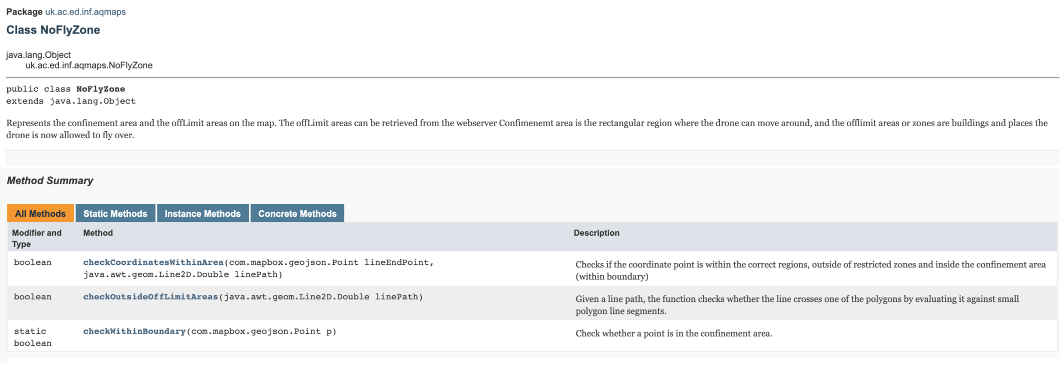
Modifier and Type	Field	Description
`private static java.net.http.HttpClient`	`client`	One HttpClient is created, shared between all HttpRequests.
`private static int`	`NOTFOUNDSTATUS`	The HTTP Response code that indicates that the server could not find the page requested.
`private static java.lang.String`	`serverName`	Current Server name is set at the beginning.

Method Summary

All Methods | Static Methods | Concrete Methods

Modifier and Type	Method	Description
`private static java.lang.String`	`changeLocationFormatToFilePath(java.lang.String w3w)`	Removes the dots in the String to a forward slash to represent the format of a file path for our webserver
`private static boolean`	`checkResponseNotFound(int status)`	Helper method that checks the HTTP Status code returned by the HTTP response.
`private static java.lang.String`	`getBodyContent(int portNumber, java.lang.String path)`	Helper function that returns the body content of a JSON file formatted as a JSON String.
`static Sensor[]`	`parseAirQualityData(int portNumber)`	Parses JSON content with air quality readings of different sensors and their location and converts them to Sensor objects.
`static NoFlyZone`	`parseNoFlyZones(int portNumber)`	Parses JSON content related to Off Limit Zones, zones inside the confinement areas that the drone should not fly over .
`static What3WordsDetails`	`parseWhat3WordsDetails(int portNumber, java.lang.String what3WordsLocation)`	Parses JSON content related to the location of a sensor and converts it to a new object of type What3WordDetails

2.8 NoFlyZone



Package uk.ac.ed.inf.aqmaps

Class NoFlyZone

java.lang.Object
    uk.ac.ed.inf.aqmaps.NoFlyZone

```
public class NoFlyZone
extends java.lang.Object
```

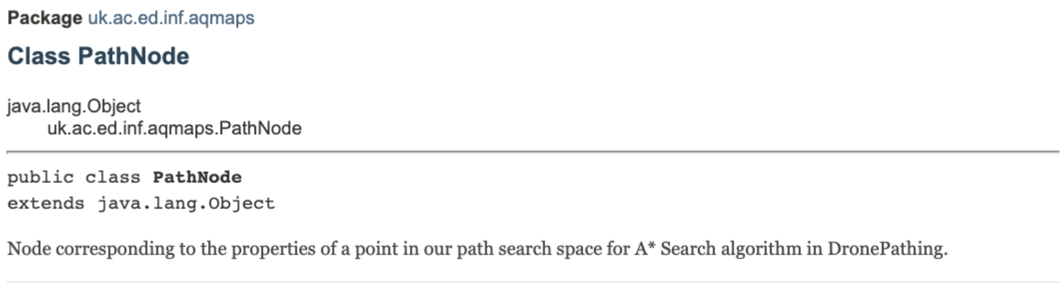
Represents the confinement area and the offLimit areas on the map. The offLimit areas can be retrieved from the webserver Confimenemt area is the rectangular region where the drone can move around, and the offlimit areas or zones are buildings and places the drone is now allowed to fly over.

Method Summary

All Methods | Static Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
`boolean`	`checkCoordinatesWithinArea(com.mapbox.geojson.Point lineEndPoint, java.awt.geom.Line2D.Double linePath)`	Checks if the coordinate point is within the correct regions, outside of restricted zones and inside the confinement area (within boundary)
`boolean`	`checkOutsideOffLimitAreas(java.awt.geom.Line2D.Double linePath)`	Given a line path, the function checks whether the line crosses one of the polygons by evaluating it against small polygon line segments.
`static boolean`	`checkWithinBoundary(com.mapbox.geojson.Point p)`	Check whether a point is in the confinement area.

2.9 PathNode



Package uk.ac.ed.inf.aqmaps

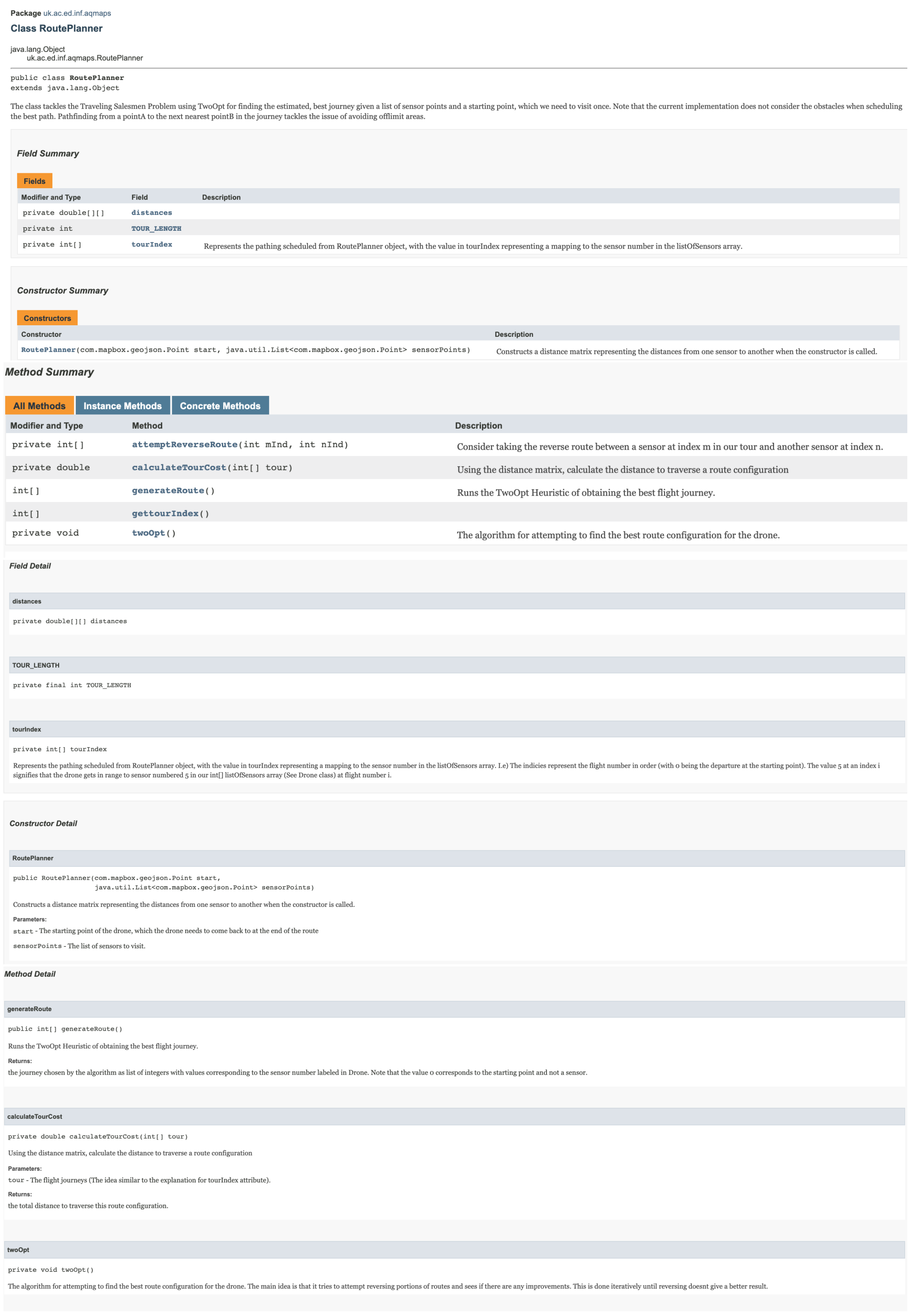
Class PathNode

java.lang.Object
    uk.ac.ed.inf.aqmaps.PathNode

```
public class PathNode
extends java.lang.Object
```

Node corresponding to the properties of a point in our path search space for A* Search algorithm in DronePathing.

2.10 RoutePlanner



Package uk.ac.ed.inf.aqmaps

Class RoutePlanner

java.lang.Object
    uk.ac.ed.inf.aqmaps.RoutePlanner

```
public class RoutePlanner
extends java.lang.Object
```

The class tackles the Traveling Salesmen Problem using TwoOpt for finding the estimated, best journey given a list of sensor points and a starting point, which we need to visit once. Note that the current implementation does not consider the obstacles when scheduling the best path. Pathfinding from a pointA to the next nearest pointB in the journey tackles the issue of avoiding offlimit areas.

Field Summary

Fields

Modifier and Type	Field	Description
`private double[][]`	`distances`	
`private int`	`TOUR_LENGTH`	
`private int[]`	`tourIndex`	Represents the pathing scheduled from RoutePlanner object, with the value in tourIndex representing a mapping to the sensor number in the listOfSensors array.

Constructor Summary

Constructors

Constructor	Description
`RoutePlanner(com.mapbox.geojson.Point start, java.util.List<com.mapbox.geojson.Point> sensorPoints)`	Constructs a distance matrix representing the distances from one sensor to another when the constructor is called.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
`private int[]`	`attemptReverseRoute(int mInd, int nInd)`	Consider taking the reverse route between a sensor at index m in our tour and another sensor at index n.
`private double`	`calculateTourCost(int[] tour)`	Using the distance matrix, calculate the distance to traverse a route configuration
`int[]`	`generateRoute()`	Runs the TwoOpt Heuristic of obtaining the best flight journey.
`int[]`	`gettourIndex()`	
`private void`	`twoOpt()`	The algorithm for attempting to find the best route configuration for the drone.

Field Detail

distances

```
private double[][] distances
```

TOUR_LENGTH

```
private final int TOUR_LENGTH
```

tourIndex

```
private int[] tourIndex
```

Represents the pathing scheduled from RoutePlanner object, with the value in tourIndex representing a mapping to the sensor number in the listOfSensors array. I.e) The indicies represent the flight number in order (with 0 being the departure at the starting point). The value 5 at an index i signifies that the drone gets in range to sensor numbered 5 in our int[] listOfSensors array (See Drone class) at flight number i.

Constructor Detail

RoutePlanner

```
public RoutePlanner(com.mapbox.geojson.Point start,
                    java.util.List<com.mapbox.geojson.Point> sensorPoints)
```

Constructs a distance matrix representing the distances from one sensor to another when the constructor is called.

Parameters:

`start` - The starting point of the drone, which the drone needs to come back to at the end of the route

`sensorPoints` - The list of sensors to visit.

Method Detail

generateRoute

```
public int[] generateRoute()
```

Runs the TwoOpt Heuristic of obtaining the best flight journey.

Returns:

the journey chosen by the algorithm as list of integers with values corresponding to the sensor number labeled in Drone. Note that the value 0 corresponds to the starting point and not a sensor.

calculateTourCost

```
private double calculateTourCost(int[] tour)
```

Using the distance matrix, calculate the distance to traverse a route configuration

Parameters:

`tour` - The flight journeys (The idea similar to the explanation for tourIndex attribute).

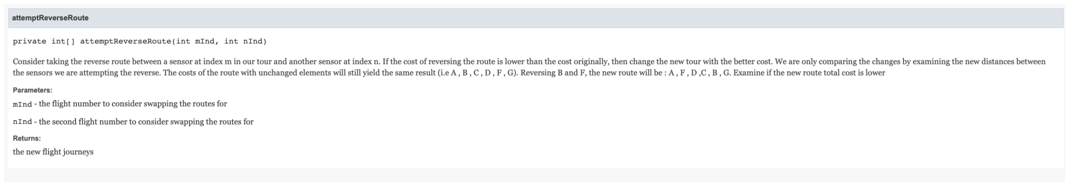
Returns:

the total distance to traverse this route configuration.

twoOpt

```
private void twoOpt()
```

The algorithm for attempting to find the best route configuration for the drone. The main idea is that it tries to attempt reversing portions of routes and sees if there are any improvements. This is done iteratively until reversing doesnt give a better result.



attemptReverseRoute

```
private int[] attemptReverseRoute(int mInd, int nInd)
```

Consider taking the reverse route between a sensor at index m in our tour and another sensor at index n. If the cost of reversing the route is lower than the cost originally, then change the new tour with the better cost. We are only comparing the changes by examining the new distances between the sensors we are attempting the reverse. The costs of the route with unchanged elements will still yield the same result (i.e A , B , C , D , F , G). Reversing B and F, the new route will be : A , F , D ,C , B , G. Examine if the new route total cost is lower

Parameters:

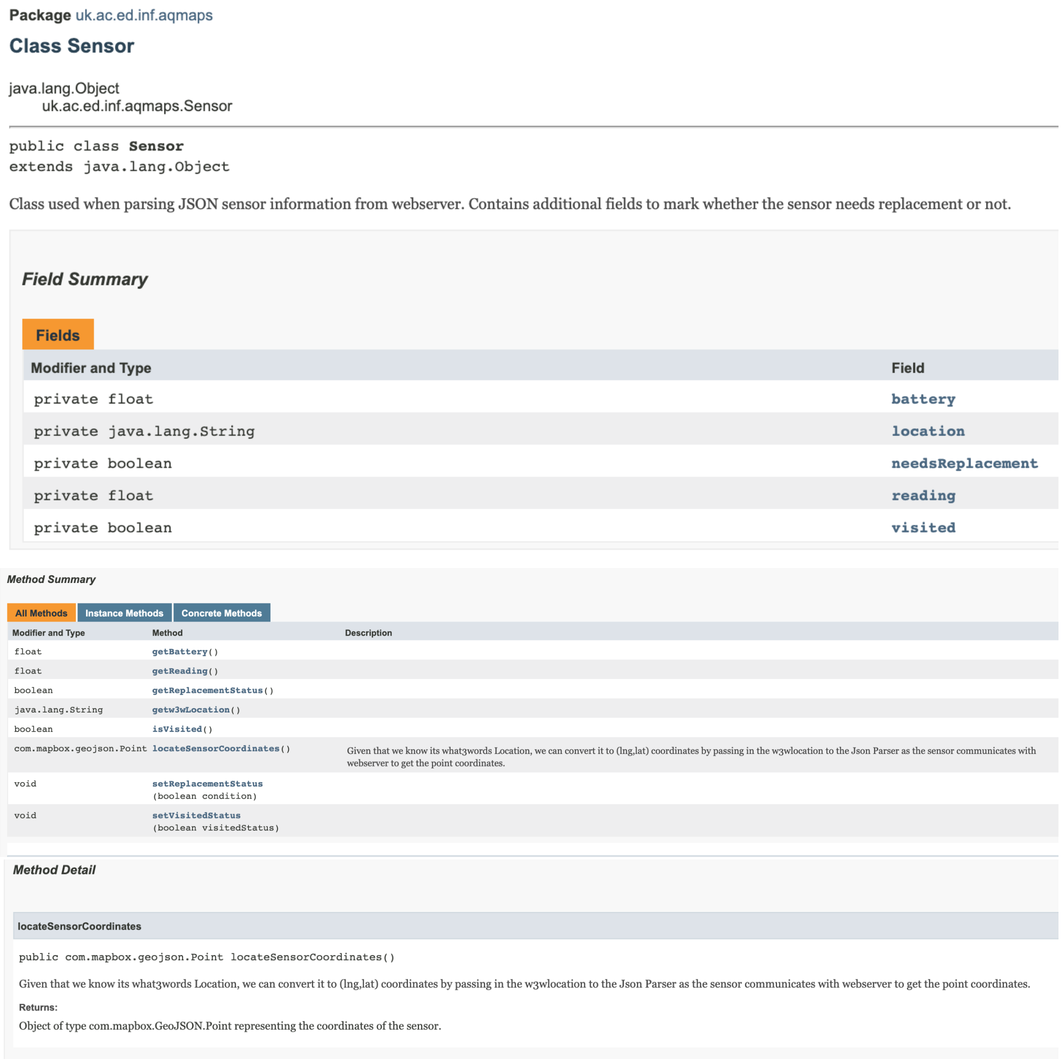
`mInd` - the flight number to consider swapping the routes for

`nInd` - the second flight number to consider swapping the routes for

Returns:

the new flight journeys

2.11 Sensor



Package uk.ac.ed.inf.aqmaps

Class Sensor

java.lang.Object
    uk.ac.ed.inf.aqmaps.Sensor

```
public class Sensor
extends java.lang.Object
```

Class used when parsing JSON sensor information from webserver. Contains additional fields to mark whether the sensor needs replacement or not.

Field Summary

Fields

Modifier and Type	Field
`private float`	`battery`
`private java.lang.String`	`location`
`private boolean`	`needsReplacement`
`private float`	`reading`
`private boolean`	`visited`

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method	Description
`float`	`getBattery()`	
`float`	`getReading()`	
`boolean`	`getReplacementStatus()`	
`java.lang.String`	`getw3wLocation()`	
`boolean`	`isVisited()`	
`com.mapbox.geojson.Point`	`locateSensorCoordinates()`	Given that we know its what3words Location, we can convert it to (lng,lat) coordinates by passing in the w3wlocation to the Json Parser as the sensor communicates with webserver to get the point coordinates.
`void`	`setReplacementStatus (boolean condition)`	
`void`	`setVisitedStatus (boolean visitedStatus)`	

Method Detail

locateSensorCoordinates

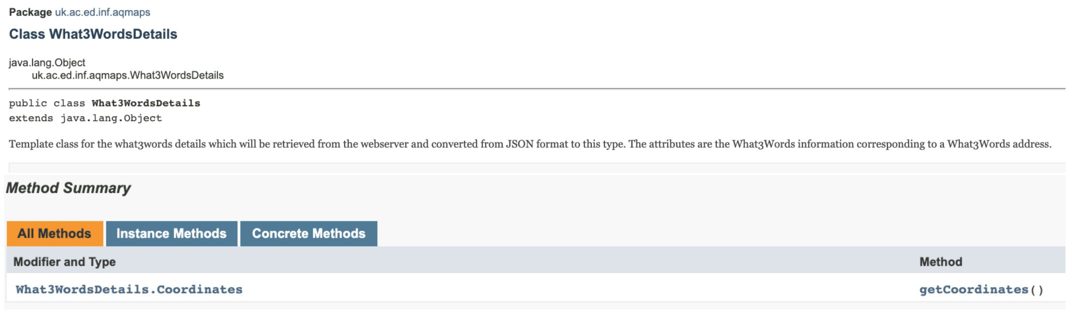
```
public com.mapbox.geojson.Point locateSensorCoordinates()
```

Given that we know its what3words Location, we can convert it to (lng,lat) coordinates by passing in the w3wlocation to the Json Parser as the sensor communicates with webserver to get the point coordinates.

Returns:

Object of type com.mapbox.GeoJSON.Point representing the coordinates of the sensor.

2.12 What3WordsDetails



Package uk.ac.ed.inf.aqmaps

Class What3WordsDetails

java.lang.Object
    uk.ac.ed.inf.aqmaps.What3WordsDetails

```
public class What3WordsDetails
extends java.lang.Object
```

Template class for the what3words details which will be retrieved from the webserver and converted from JSON format to this type. The attributes are the What3Words information corresponding to a What3Words address.

Method Summary

All Methods | Instance Methods | Concrete Methods

Modifier and Type	Method
`What3WordsDetails.Coordinates`	`getCoordinates()`

3 Drone Control Algorithm

This section describes the algorithms used to allow the drone to navigate to all the sensors whilst avoiding the no fly zones in its confinement area.

In order to tackle this, the problem was split into two sub-problems: A Traveling Salesmen Problem (TSP) and a PathFinding Problem. There are many available algorithms to tackle each of these, and the ones used in the software were Two-Opt and A* Search. The report will go into more details about the descriptions of each.

3.1 Route Planner-Solving Traveling Salesmen Problem Using Two-Opt

3.1.1 Algorithm Description

Given that we have a list of sensors and their coordinates, our main concern here is to try to find the shortest route from the starting point to all sensors and back to that point at the end (within 150 moves if possible). To find the best tour for our drone and visit each sensor once, we need to order the trip such that the total displacement for the drone movement is the least (we want to aim for traversing with as little distance as possible). We will approximate that route by using pairwise exchange, or Two-Opt, which is a local search heuristic method.

The algorithm starts off by computing the distance (adjacency) matrix, where the value at (row i, column j) represents the distance (edge cost) from point i to point j. Then, the process includes iteratively picking two points on our whole tour journey and reversing the route in the segment in between them.
To further illustrate, if the original route was
A - B - C - D - E - F - G,
reversing two arbitrary points B and F, the new route will look like :
A - F - E - D - C - B - G.

The algorithm then computes the total distance of the original route and the new route and compares them with each other. It decides to choose the new route if there's an improvement, meaning the cost of the total route distance for the new route is the lesser one. Otherwise, it stops once there is no further improvement on reversing a route.

Psuedocode for Two-Opt:

```
improved = True;
while loop until no improvement is found{
    //Assume there is no improvement and we already have the best tour path with the minimal distance
    improved = False;
    // Calculate current tour cost;
    minimalDistance = calculateTourCost(currentTourRoute)

    for (m = 1; m <= number of points to consider; m++) {
        for (n = m + 1; n <= number of points to consider ; n++) {
            newRoute = attemptReversingRoute(currentTourRoute, m, n)
            newDistance = calculateTourCost(newRoute)
            if (newDistance < minimalDistance) {
                existing_route = newRoute
                minimalDistance = newDistance
                //We found a better path, we will have to loop again.
                improved = True
            }
        }
    }
}
```

3.2 DronePathing - PathFinding Algorithm Using Weighted A* Search

3.2.1 Algorithm Description

Once the program produces the estimated best route for the drone using **RoutePlanner**, the **DronePathing** class handles a different problem: navigating from one route point to the next one.

Given we have two points on a map and 36 possible directions(angles in multiples of 10 from 0 to 350 inclusive), our sub-problem here is to navigate our robot to reach the next destination point from a point in our route whilst trying to avoid obstacles, the off limit areas (and to avoid going out of bounds too). Hence, this task can be reduced to a graph search one, the algorithm used is A* Search.

To further illustrate the end goal of this algorithm, consider the figure below

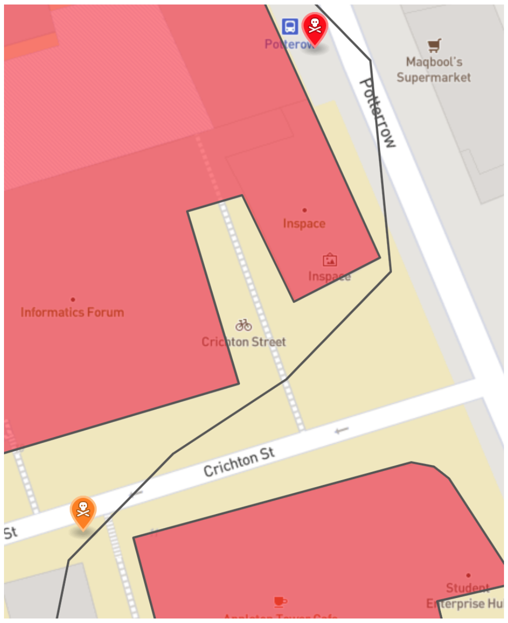


Figure 4: A Star Search - The drone navigates from a point to another to reach another sensor by avoiding the red marked zones. Date: 21/04/2021, Coordinates: Latitude of 55.942686 and Longitude of -3.191009

The algorithm makes use of **PathNode** objects in the middle of the path finding algorithm to keep information on its current attributes and on its parent **PathNode** in the graph search space (null parent meaning the current node is root). This **PathNode** class allows us to maintain a tree of paths from the start node (created with the point coordinates given to the generatePath() algorithm) and extending those paths one move at a time until the target point is in range with the sensor reading. The attributes saved include its cost score so far, its Point coordinates, the number of moves to reach the current node in space, and the angle set later on which points to the direction of the next node created.

The algorithm generates potential nodes which are pathing that it can take (36 possible directions). Once generated, in the next iteration, it tries to determine which of the paths from those nodes to expand and try to traverse to, choosing the node that minimizes this function:
f(n)=g(n)+h(n)
where g(n) is the current cost from start node to the node n (the displacement moved by the robot so far), h(n) is the heuristic function which estimates distance from current node n to the target node (which is simply the euclidean distance here), and f(n) is the summation of those scores.

Note: This turned out time-consuming as the algorithm generated many nodes. Hence, in our version of A* Search, the process of finding the node is sped up at the potential slight cost of being optimal by multiplying the heuristic function with an epsilon cost of 1.5 : h(n) * E, where E=1.5. This allows the algorithm to consider expanding fewer nodes because the heuristic function is given more importance (more weight is given to nodes closer to the goal than those away). In addition, for the current implementation, if the starting point is within the acceptable range to the end point, the algorithm forces a movement in order to replicate the lifecycle of the drone (in that it has to take a movement first before taking the reading).

Simplified Pseudocode:

```
generatePathAStar(Point  startCoords,  Point  targetCoords)  {

//open consists of nodes that have been visited but not expanded yet.
//close consists of nodes that have been visited and expanded.
open  =  new  ArrayList<PathNode>();
closed  =  new  ArrayList<PathNode>();
foundPathSuccessfully  =  false;
//create a new node and add it to the open list
startNode  =  new  PathNode(startCoords);
open.add(startNode);
while(open is not empty)  {
    //Find the node with the best Fscore
    for  (node in open)  {
        //f(n) = g(n) + h(n)
        totalFScore  =  node.getCostG()  +
        heuristicFunction(node.getPointCoordinates(),  targetCoords);
    }
    open.remove(bestNode)
    closed.add(0,bestNode);

    if  (target criteria is met)  {
        finalNode = bestNode
        //we found a solution
        foundPathSuccessfully=  true;
        backTrackAndreturnThePathGenerated(finalNode)
        break;
    }

    //If we havent found solution yet, find the nodes next to the current node to consider
    //to expand the route for.
    nextPossibleNeighbours=  findNextPossibleNodes(currNode);
    for  (neighbour in nextPossibleNeighbours)  {
        //If the neighbour does not belong in the open and closed list, then we can add it to
        the open list to consider expanding on in next iteration.
        if  (!searchNeighborInList(open,neighbourNode)  &&
        !(searchNeighborInList(closed,neighbourNode))){
        open.add(0,neighbourNode);
            }
        }
    }
```

3.3 Combining both Algorithms- End Result

The figure below is the readings output file generated by the program at the end.

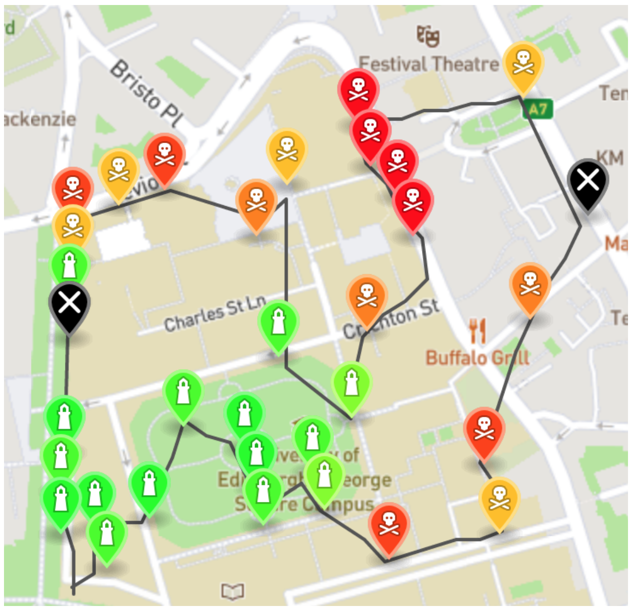


Figure 5: End Result - The red zones were removed to match up with the readings output format of what the readings produce from the program after the path is computed. Date: 21/04/2021, Coordinates: Latitude of 55.942686 and Longitude of -3.191009