The University Of Edinburgh

Pablo Miró Ruiz

s1814033

Informatics Large Practical (2020 - 2021)

Author
Miro Ruiz, Pablo

Professors
Prof. Gilmore Stephen
Prof. Jackson Paul

December 4, 2020

Contents

1 Introduction

The purpose of this practical is to program an autonomous drone which will collect readings from air quality sensors distributed around the University of Edinburgh's George Square campus as part of a fictitious research project to analyse urban air quality.
Furthermore, scientific visualizations will be produced so that the results can be understood without getting to know the implementation of the algorithm. This would make it easy for the fictitious researchers apply for more funding in order to develop a large-scale version of the project.

2 Drone control algorithm

2.1 Introduction to the algorithm

Having stated the task, I will start by examining the approach I used to solve this problem. I believe this will help you, the reader, understand why I made several decisions regarding the software architecture which will be discussed later.
I divided the algorithm of this problem in two parts: the **planning** and the **execution** phases. The main reason why I decided to split the algorithm into such parts was because it has been proved as a great way to enhance computational complexity in this kind of problems. The main idea behind this is the famous *Traveling Salesman Problem*, on which I made a report last year (find it here) evaluating several of the approximation methods (also known as heuristics) which could be used to solve this in an efficient way.
Taking inspiration in my last year's report, I decided to use reduction so that I could model the given drone as a "salesman" trying to visit a given set of sensors instead of cities. It is worth mentioning that in the case of the drone, the weight function we would use for an edge will be the Euclidean distance between the two nodes making up that given edge.
Consequently from this reduction, since the TSP belongs to the set of NP-Complete problems, we must rely on heuristics or methods which try to optimize as much as possible the solution using some approximation. There is a great deal of information in **CLRS** sections 34 & 35 talking about NP-Completeness and Approximation Algorithms which turned out to be super helpful. If you would like to know more about this, I would also recommend you looking at the report I wrote since I also make references to CLRS and explain why some approaches are better than the others apart from implementing them in Python.

2.2 Reduction to a graph problem

As mentioned previously, I decided to model this problem as a graph. However, given the small scale of the system and the resources available I felt it was not necessary to build the connected graph explicitly with 34 undirected edges. The reason was the fact that I decided to follow a Greedy approach and before executing the algorithm I could get the order in which the drone would visit the sensors first. For future references, given a larger scale and different zones it might be a good idea to try out several other approaches such as building a minimal spanning tree using either Prim's or Kruskal's algorithms. I will leave this to future scientists.

2.3 Greedy planning stage

A Greedy algorithm is an algorithm which builds up the end solution piece by piece, always choosing the most optimal solution locally, i.e. taking into account only the current position of the drone (for the specifics of our given problem).

The approach we follow in the planning stage can be easily described as a **Nearest neighbor search (NNS)**, which is a form of proximity search. To sum up, it consists on finding the point in a given set that is closest (or most similar) to a given point. Closeness is typically expressed in terms of a dissimilarity function: the less similar the objects, the larger the function values. In our case, the dissimilarity function is defined as the Euclidean distance between those two points being taken into consideration.
The planning executing is somewhat simple and the idea behind is to set up everything we will need:

1. Set up the project: Obtain the 33 sensors which the drone should visit, the *no-fly-zones* which it must not go through and the location from which the drone starts.

2. Iterate through the given sensors and order them in terms of the Euclidean distance: take the current location and find the closest sensor which will be added to the list. Update **currentLocation** to the one from the last sensor being added.

3. Perform step 2 until there are more sensors to order.

This is all done by the method **setUpMap** from the **BuildAqmap** class which retrieves the data by using GET requests from the **ServerRequest** class and orders the sensors by using **getSensorsToVisitInOrder** from **Utils** class. We will discuss how these classes interact amongst one another in the next section.
In the following diagram you can easily get a general idea of the methods and fields used by the afforementioned classes to implement the set up:

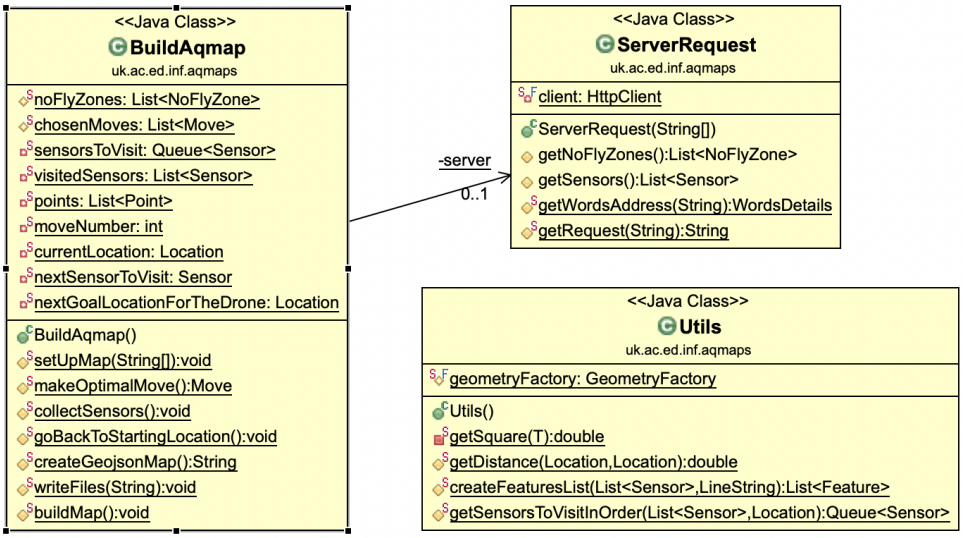

Figure 1: Classes involved in the set up of the project

2.4 Greedy execution stage

In this subsection I will discuss how the algorithm works after having set up the project successfully. As you can expect, this is the main core part of the algorithm which in turn makes it also a key part of the overall practical.
It makes sense to point out the constraints the drone is dealing with so that you can understand why do some methods exists:

- The drone has a maximum of 150 moves; each with a defined length of 0.0003 degrees.
- The drone cannot go through any of the zones determined as *no-fly-zones*.
- The drone can only rotate in an angle which is a multiple of 10 degrees.
- The drone has a life cycle pattern: make a move, check for sensor.
- The path performed by the drone should be a closed loop as near as possible.

The algorithm is implemented in **BuildAqmap** class and uses help of several methods from an auxiliary class, namely **BuildAqmapUtils**. Having set up the project, the **buildMap** method tries to collect readings from every sensor in a step by step procedure: This was approached by keeping track of two things: the current location of the drone and the sensor on which the drone must focus. Then, the algorithm calculates which is the most optimal move (i.e. the one which does not cross any *no-fly-zone*, has an angle being multiple of 10 and does not go out of the confined area). Once this move has been performed, the drone checks if the sensor is in range: if it is not, the drone keeps moving until it gets close enough to the sensor. Otherwise, it collects its reading and updates the location on which it is focusing on (**nextSensorToVisit** and **nextGoalLocationForTheDrone**). In the case where there are no more sensors to visit, the drone will focus on the starting position (after running the **collectSensors()** method we run **goBackToStartingLocation()**) as long as it has performed less than 150 moves. One nice idea I want to point out is the fact that when the drone is looking for the possible move it can make next, it also takes into account not going to a previous location where it has already been (using a radius which has been determined by trial and error). I decided to add this check because sometimes the drone will get stuck and would go back and forward spending all of its moves without doing anything useful. However, it is to mention that I also considered the case when the drone enters a dead end: in this case we do not remove those moves that go back to a location where the drone has already been.
The algorithm relies in several auxiliary methods which are used in **BuildAqmap** and can be found in the **BuildAqmapUtils** class. I will discuss here the most important ones.

`getPossibleMoves` is a method which returns a list of tuples (defines as Java Pair(s)) which the drone could perform given the location on which it stands.

`isInConfinedArea` is an auxiliary private method which ensures a given point (specified as a member of the Location class) can be found inside the confined area. Returns true if the point is inside; false otherwise.

`doesIntersectWithNoFlyZones` is an auxiliary private method which ensures that the line connecting two given locations does not intersect with any of the *no-fly-zones* of the map.

`isGoingToPreviousPosition` is an auxiliary method which ensures a given point (specified as a member of the Location class) can be found inside the confined area. Returns true if the point is inside; false otherwise.

`filterPossibleMoves` is a method which removes those moves which will make the drone leave the confined area or cross any of the *no-fly-zones*.

The key idea of the algorithm uses the method `makeGreedyMove()` which is taking into account the location to which the drone is moving. Moreover, it makes use of several methods which represent the life cycle of the drone: (1) make a move, (2) collect sensor's reading as long as the sensor is close enough. If the sensor is not close enough the drone will keep on making greedy moves; otherwise it will update the location on which it will be focusing on.

2.5 Testing

I decided to write a python3 program which went over every possible day from the Web server and checked several things for the drone:

- if it spent every move before going to the origin where it started.
- if it produced inconsistent results skipping moves or using invalid angles.
- if it left the confined area.
- if it entered any *no-fly-zone*.

Thanks to this testing I realized that in some days the drone got stuck and entered an infinite loop spending moves without making any progress. That is why I added the method `isGoingToPreviousPosition` and it has produced correct results since.

Before moving on to the next section, in the following page I show two of the maps produced by the drone in two different days.

Figure 2: Rendered map on March 27th 2020

Figure 3: Rendered map on December 28th 2021

3 Software architecture description

General overview

I will use this section to give a general view to the way I divided my classes for this project. I will start mentioning the classes which I used to retrieve information from the command line and from the web server. Following this, I will end up talking about the classes which go through the algorithm planning and executing phases as I outlined briefly int he previous section. The project can be seen as split conceptually in three different sections: `Model Section`, `Algorithm Section` and `Map Section`.

3.1 Model Section

This part of the project deals with parsing the input from the command line, producing the correct output mentioned in the coursework specification and making requests to the server to retrieve the required information. It also includes given constants from the coursework specification which can be easily replaced in case they change. The classes which I consider to do this are `Date.java`, `Constants.java IO.java` and `ServerRequest.java`. Furthermore, I have also included the data structures which directly correspond to the mapping between the JSON and GeoJson objects from the server and the Java classes, i.e. the deserialization process: `WordsDetails.java`, `Sensor.java` and one class which lets us consider each of the GeoJson *no-fly-zones* as separate instances of the `NoFlyZone.java` class.

`Constants.java` provides access to the constraints provided in the coursework specification. It makes it easy to access and change them were it necessary.

`Date.java` This class makes it easy to keep the date provided as a command line argument to be used in the future and to format it with $'0'(s)$ in front if *day* or *month* are < 10 .

`IO.java` The IO class provides any method needed to parse the command line arguments and to write the required files. The output format can be easily accessed and changed from here.

`ServerRequest.java` is in control of the GET requests done in the server to retrieve information about the *no-fly-zones*, sensors and the corresponding address to a given *What3Words* address.

`NoFlyZone.java` This class lets me access every *no-fly-zone* so that I can convert it to a GeoJson Polygon in case I want to visualize the zone or I can convert it to a JTS Polygon in order to use this library's methods which ended up being really useful to check for intersections, for example.

`WordsDetails.java` lets me deserialize the *What3Words* address which can be accessed from the server as a JSON object. This class is here so that from a given *What3Words* address I can access the specific coordinates to which they refer.

`Sensor.java` Similarly to the *WordsDetails.java*, this class is used to deserialize the list of 33 sensors available on a given day. Furthermore, it provides functionality to retrieve the position of the sensor as an instance member of our Location class as well as making it a GeoJson feature so that it can be rendered. In case we would like to change the sensor's properties (colour, symbol...) this can be easily modified from here.

3.2 Map Section

Speaking about this **Map layer**, this is the one which implements the way in which the sensors are supposed to be displayed in the map. I also took into consideration what would happen if some sensors have not been visited by the drone (they are coloured in grey and without any symbol). Moreover, I defined a separate class, namely `Location.java`, which let me modify the way I represented a given point in the map so that I could modify the given point to represent it visually using GeoJson, or I could modify it so that I could use the Java Topology Suite which helped me check intersections with the *no-fly-zones* and if the given location was found inside the confined area. And last but not least, it can be found a class representing each move made by the drone.
By doing so, it will be very convenient for future programmers to change the visualization of the sensors in the map or the way the drone is allowed to move.

`Move.java` Every move of the drone is modeled as an instance of this given class. It makes it really intuitive to keep a history of the drone's moves containing both the start and end locations plus the direction in which the drone was moving. If the drone could take a reading from a given sensor after performing this move, that sensor will be associated with the move.

`Marker.java` provides methods to get the corresponding colour and symbol which corresponds to a particular sensor.

`Location.java` This is probably one of the most useful classes since it gives us the possibility of changing a Point between two libraries used in the project, depending on whether we are interested on visualizing a given Point or we want to check if that point is inside the confined area, the path to which it belongs does not intersect a *no-fly-zone*, etc.

3.3 Algorithm Section

This is the main part of the practical because it contains the implementation of the Greedy algorithm used to control the drone. It has already been discussed in section 2.

`Utils.java` provides general methods which will be used in the program: getting the Euclidean distance between two points, defined as:

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (p_i - q_i)^2} \tag{1}$$

where $n = 2$ for our 2D modelling. Furthermore, this class provides a method which lets us create the list of features from the given sensors and line string which we can use to create a feature collection. And last but not lest, it lets us perform one part of the planning stage of the algorithm: getting the sensors to visit in order and returning them using Java's Queue interface which provides $O(1)$ time to retrieve the head.

`BuildAqmapUtils.java` As the name of the class says, it provides functionality which will be useful to build the air quality map. This contains the key methods to obtain the possible moves and filter them according to the constraints we have using `filterPossibleMoves` and then getting the closest one to the location to which the drone is moving to with `getOptimalMove`

`BuildAqmap.java` Our key class for the algorithms, this performs the life cycle of the drone making it collecting the sensors and going back to the previous positions move by move (in a greedy manner as we mentioned). It also allows us to create the corresponding GeoJson map to visualize the path performed by the drone as well as writing the files required in the coursework. And last but not least, this class is the one which sets the project up and keeps track of the moves done by the drone and of the points and sensors visited by it.

3.4 Software Architecture Diagram

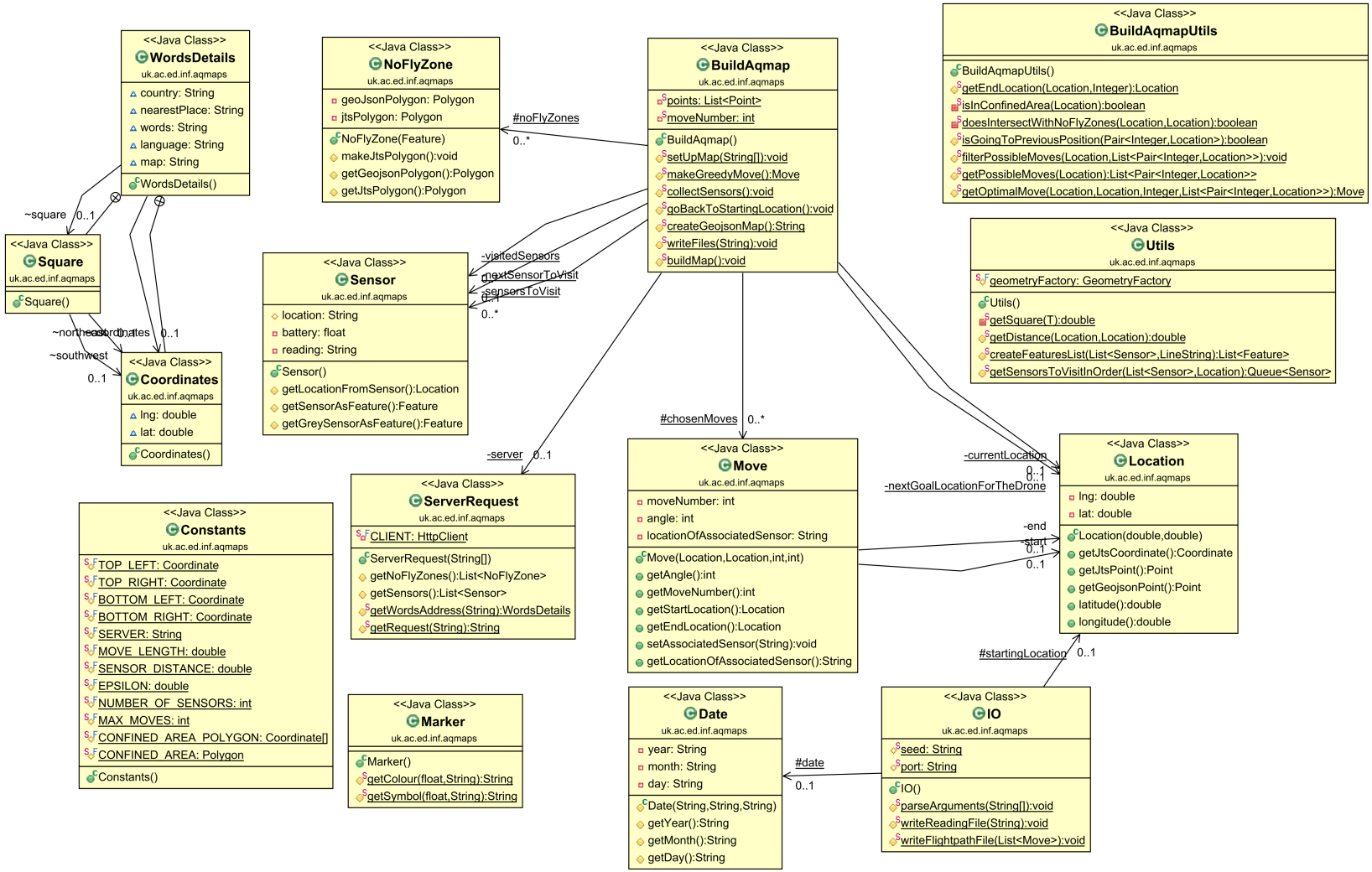

Figure 4: Complete Software Architecture Diagram

4 Class documentation

Finally, I will leave this section for the documentation of the most important classes.

4.1 IO.java

protected static fields: `Date date`, `Location startingLocation`, `String seed`, `String port`

methods:

`protected static void parseArguments(String args[])`
Sets up the date, starting location, seed and port fields from the arguments provided.

`protected static void writeReadingFile(String stringToWrite)`
Creates the readings file corresponding to the previously parsed date and writes to it the JSON string representing the air quality map.

`protected static void writeFlightpathFile(List<Move> moves)`
Creates the flightpath file corresponding to the previously parsed date and writes to it every single move the drone performaed following the format given in the coursework specification.

4.2 ServerRequest.java

private static final field: `HttpClient CLIENT`

constructor: `ServerRequest (String[] args)`
Use constructor to parse command line arguments before making any request to the server.

methods:

`protected List<NoFlyZone> getNoFlyZones() throws InterruptedException`
Returns list of NoFlyZone objects representing each of the *no-fly-zones* obtained from the server.

`protected List<Sensor> getSensors() throws InterruptedException`
Returns list of sensors to be visited by the drone on the date parsed from the command line.

`protected static WordDetails getWordsAddress(String words) throws InterruptedException`
Returns a WordsDetails object representing the address of the What3Words string given as argument after its deserialization.

`protected List<Sensor> getRequest() throws InterruptedException`
Performs a GET request with the given argument as the path connecting to the sever in the port provided as command line argument. Returns the result as a String.

4.3 NoFlyZone.java

private fields: `com.mapbox.geojson.Polygon geoJsonPolygon`, `org.locationtech.jts.geom.Polygon jtsPolygon`

constructor: `NoFlyZone (Feature feature)`

methods:

`protected void makeJtsPolygon()`
Method to convert the GeoJson Polygon representing the *no-fly-zone* to a JTS Polygon.

`protected com.mapbox.geojson.Polygon getGeojsonPolygon()`

`protected org.locationtech.jts.geom.Polygon getJtsPolygon()`

4.4 Sensor.java

fields: `String location`, `float battery`, `String reading`

methods:

`protected Location getLocationFromSensor() throws InterruptedException`
Returns the location of the sensor using the field containing the What3Words encoding after parsing it in order to get the location as an instance of our Location class.

`protected Feature getSensorAsFeature() throws InterruptedException`
Returns the feature corresponding to the instance of the Sensor class. It also adds the five properties that feature should have according to the coursework specification.

`protected Feature getGreySensorAsFeature() throws InterruptedException`
Returns the feature corresponding to the instance of the Sensor class if it has not been visited so that it has a grey colour and no symbol.

4.5 Move.java

fields: `Location start`, `Location end`, `int moveNumber`, `int angle`, `String locationOfAssociatedSensor`

constructor: `Move (Location start, Location end, int number, int angle)`

methods: (pretty self-explanatory)

`protected int getAngle()`

`protected int getMoveNumber()`

`protected Location getStartLocation()`

`protected Location getEndLocation()`

`protected String getLocationOfAssociatedSensor()`

`protected void setAssociatedSensor(String location)`

4.6 Marker.java

methods:

`protected static String getColour(float batteryLevel, String reading)`
Returns a String which represents the colour the sensor will have with if its properties are given by the arguments.

`protected static String getSymbol(float batteryLevel, String, reading)`
Returns a String which represents the name of the symbol to use for a Sensor which has the given battery level and reading.

4.7 Location.java

private fields: `double lng`, `double lat`

constructor: `Location (double lat, double lng)`

methods:

`protected Coordinate getJtsCoordinate()`

Returns a Coordinate object using the class' fields so that the Java Topology Suite can be used with the Location specified.

`protected org.locationtech.jts.geom.Point getjtsPoint()`
Returns a JTS point using the fields from the instance of the given class as latitude and longitude.

`protected com.mapbox.geojson.Point getGeoJsonPoint()`
Returns a GeoJson point using the coordinates from the class' fields.

4.8 Utils.java

protected static final field: `GeometryFactory geometryFactory`

methods:

`private static <T extends Number> double getSquare(T n)`

`protected static <T extends Number> double getDistance(Location p1, Location p2)`
Returnt the Euclidean distance between the two given locations as arguments of the method.

`protected static List<Feature> createFeaturesList(List<Sensor> sensors, LineString lineString) throws InterruptedException`
Returns a list of features made up of the features corresponding to every sensor (given as an instance of Sensor class) which is going to be visited by the drone and the feature corresponding to the path made by the drone given as a LineString object.

4.9 BuildAqmapUtils.java

methods:

`protected static Location getEndLocation(Locatin start, Integer angle)`
Returns the location in which the drone will be if it moved with a given direction starting in the given location.

`private static boolean isInConfinedArrea(Location location)`
Checks if a specific point (given as an instance of the Location class) can be found inside the confined area extracted from the server.

`private static boolean doesIntersectWithNoFlyZones(Location start, Location end)`
Checks if the line starting at "start" and finishing at "end" intersects with any of the no fly zones so that we know the drone cannot go from that start location to the end location given as arguments.

`private static boolean isGoingToPreviousPosition(Pair<Integer, Location> possibleMove)`
Checks if the drone is going close to any of the previous moves it has already performed.

`protected static void filterPossibleMoves(Location start, List<Pair<Integer, Location>> possibleMoves)`
Method which filters the possible moves the drone could do using our other methods to check that those moves do not intersect any no fly zone and they are inside the confined area.

`protected static List<Pair<Integer, Location>> getPossibleMoves(Location start)`
Returns every possible move (as a Pair¡Integer, Location¿) the drone could perform from a given location.

`protected static Move getOptimalMove(Location moveStartLocation, Location nextSensorLocation, Integer moveNumber, List<Pair<Integer, Location>> filteredPossibleMoves)`

Using the possible moves the drone could make, it returns the one which gets closer to the location the drone has as a target.

4.10 BuildAqmap.java

protected static fields: `List<NoFlyZone> noFlyZones`, `List<Move> chosenMoves`

private static fields: `ServerRequest server`, `Queue<Sensor> sensorsToVisit`, `List<Sensor> visitedSensors`, `List<Point> points`, `int moveNumber`, `Location currentLocation`, `Sensor nextSensorLocation`, `Location nextGoalLocationForTheDrone`

methods:

`protected static void setUpMap(String[] args)`
Sets the server up so that we retrieve the sensors, the no fly zones and initialises the variables we will use later.

`protected static Move makeGreedyMove()`
Returns the most optimal move for the drone to take so that it gets closer to the next sensor (or starting position) taking into consideration the constraints (no fly zones and confined area). It also takes into account the previous moves so that the drone does not go back and forth to the same place wasting its moves. However, if the only possibility is going back, it will do so since there are some special cases which have to be taken into account (imagine going to a zone which has the same place to enter and exit.

`protected static void collectSensors() throws InterruptedException`
makes the drone go from one sensor to the next as long as it has enough moves or it has not finished checking every sensor retrieved from the sever.

`protected static void goBackToStartingLocation()`
If the drone has not done the maximum number of moves it can do, try to make it go back to the origin. We do so in a similar way to how we go to the sensors, but this time we consider a different length from the origin (Constants.MOVE_LENGTH instead of Constants.SENSOR_DISTANCE) to finish.

`protected static String createGeojsonMap() throws InterruptedException`
Returns the corresponding GeoJson map taking into account the points through which the drone has gone through (so that we can render the path as a LineString), the sensors it has visited and also the ones it has not visited (which will be grey-coloured and without any symbol.

`protected static String createGeojsonMap() throws InterruptedExceptionn`
Returns the corresponding GeoJson map taking into account the points through which the drone has gone through (so that we can render the path as a LineString), the sensors it has visited and also the ones it has not visited (which will be grey-coloured and without any symbol.

`protected static void writeFiles(String jsonString)`
Writes the two requested files in the coursework.

`protected static void buildMap() throws InterruptedException`
Method which completes the map collection the sensors and trying to go back to the origin using the methods previously defined.