Informatics Large Practical
Coursework 2

Weiyu Tu
s1820742

Contents

Software Architecture Design

Class Documentation

Algorithm Explanation

Software Architecture Design

Introduction

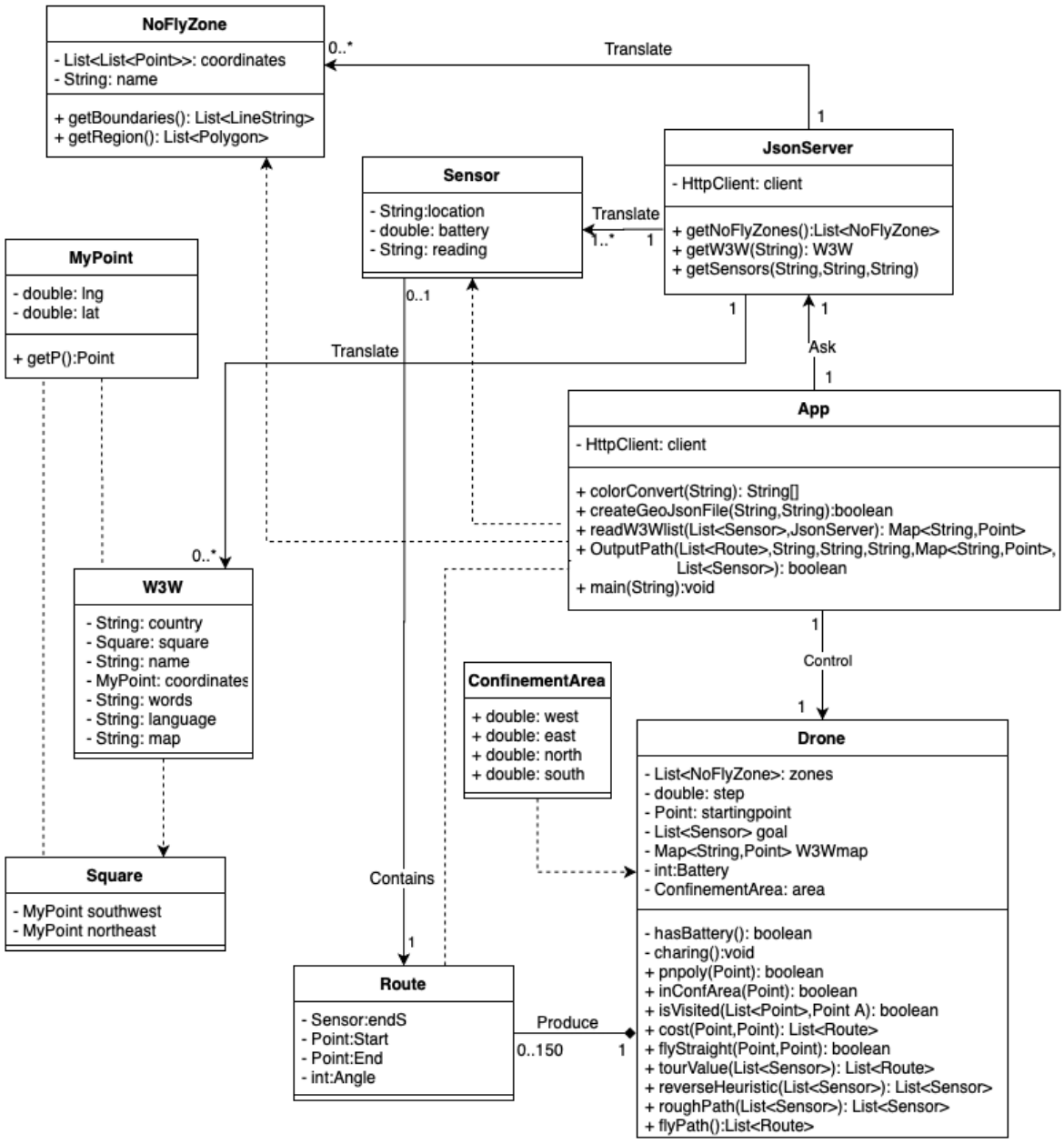
The Application is used for an autonomous drone which its task is to take readings from the air quality monitoring sensors which are distributed irregularly in the central location of the University of Edinburgh. The input argument should be in the form of day+ month+ year+ starting point latitude+ starting point longitude+ random seed+ port number, e.g.:

```
15 06 2021 55.9444 -3.1878 5678 80
```

The two output files will be 1. Air-quality-readings geojson file and 2. Flight-path text file, e.g.:

`Flightpath-15-06-2021.txt` and `readings-15-06-2021.geojson`

UML Class Diagram



Description

The entrance of the Application is in the ***App*** class. ***App*** class has the main method to receive the input argument such as the date, the port and the starting point. It is also the exit of the Application since it generate the output files: flightpath txt file and the readings geoJson file.

—APP

We need to translate filename which composited by the input arguments into required information about the air quality sensors which are waiting to be read and the no fly area. The provided no-fly-zone maps, the daily readings and the words-represented location files should be read, translated and saved by the ***JsonServer*** class. It handles Application's networking connecting operations and translates the web files into String object, which finally saved as required type and returned.

—JsonServer

As mentioned above there are 3 kinds of files needed to be translated by the ***JsonServer*** :

NoFlyZone should be generated through the webServer from a *no-fly-zones.geojson* file. Each object represents a close edged zone on the map which has a List of List of vertices and the zones's name as attributes. The drone can not pass through any ***NoFlyZone***. The drone can only fly to the other side on the zone by flying around the buildings. Since there might be more than one ***NoFlyZone*** on the map, JsonServer will return a List of ***NoFlyZone*** to the main ***App*** class.

—NoFlyZoner

Sensor is the class which represent the real world sensor and carries the related information e.g.location, reading, battery. The ***JsonServer*** will read local map*/yy/mm/dd/readings.Json* files. If there are 34 sensors listed in the Json file then ***JsonServer*** will read and return a list of sensor which contains 34 ***Sensor*** instance. The list will be parsed back into the main ***App*** class. However, till now we still can not get the real location of each sensor. The location of the sensors now are represented in the form of "word1.word2.word3" which is the String. If there is no ***Sensor*** in the list, the ***App*** will regard it as an exception which means you had better not assigned no work to my little drone.

—Sensor

Then I create a ***W3W*** class. Given a "word1.word2.word3" String as the input parameter, the ***JsonServer*** will return a ***W3W*** instance. ***JsonServer*** reads the *words/word1/word2/word3/details.Json* and translate the web content into String and finally to ***W3W***. Then we can know the its location in the form of longitude and latitude. As mentioned above, there is more than one ***Sensor*** that needed to be read and every ***Sensor*** has its ***W3W*** location. It will return one ***W3W*** every time you call the function but can deal with all the ***W3W***s .

—W3W

What's more, there are two classes ***MyPoint*** and ***Square*** which are helper classes facilitate the parsing process.

—MyPoint / Square

After we had collected all the required information or a flight, we knew where can not fly through and which sensors should be read together with their locations, we can start the ***Drone***. All the movements predicting, sensor's permutations planning and obstacle avoiding detecting are done by the methods in the ***Drone*** class. ***App*** will initialise one ***Drone*** each time and once we call the ***Drone***'s flyPath() method, the embedded *flying with obstacle avoiding* algorithm will be triggered. we can get drone's flight path.

—Drone

The flight path is represented by a list of ***Route*** class. A ***Route*** carries the information of where it the flight starts and ends, the angle of the path (0° means towards the East) and the sensor that it has reached and read. In another word, one ***Route*** instance stands for one step of the drone. It is 0.0003 degrees from the ***Route***.start to the ***Route***.end.

—Route

In addition, the ***ConfinementArea*** is also a separate class which provide the edge of the map. The ***ConfinementArea*** instance is initialised in the ***Drone*** class and we only have one fixed map for this application.

—ConfinementArea

Most of the methods are public. Most of the class attributes are private. I have generate getter and setter functions for those attributes. Some of them are final. Details will be mentioned in the following Class document.

Class Documentation

There are 10 classes in total.

#1 App.java

It is the main controlling class of the Application or we can regard it as the trigger of all the classes to be activated. The class only has one private static final attribute:

- The attribute `client` is of type HttpClient and be initialised as it is defined. It is private because It is not going to be used by other classes. It is static and final because it will not be replaced once it is generated.

The class also has 5 public methods:

- The `colourConvert ( double battery, String reading )` method. As we got the Sensor's battery readings and air-quality readings, we put the readings into the function and it will return a String[ ] of size 2 which the first String is a RGB String and the second is the marker symbol e.g. "lighthouse", "danger", "cross". All the deterministic process are supported by if-else function. The first thing is to check whether the battery is powered off or of low power. If the battery reading is less than 10, the function will return a black colour RGB String and "cross" as the marker symbol. Also, if the air-quality reading itself is "NaN" or "null", the function will return the same thing as above. A cross symbol with black marker tells us there is no air-quality reading for this Sensor. Then if the Sensor has enough battery, we can translate the readings to colourful string but if-condition. The smaller the air-quality reading, the fresher the air, the marker is more green as red represents the air-quality is super bad.

- The `createJsonFile ( String jsonString, String fileName )` method which save a jsonString into a geoJson file. I initialise a File instance and a Writer instance to write the file. It just follows the normal write file process as writing txt file which only change the file suffix from txt to geoJson.

- The `readW3Wlist ( List<Sensor> list, JsonServer server )` method. Since I will repeatedly look up the real world coordinates of each Sensor, it is more convenient that just generate an address book which mapping the "word1.word2.word3" String with a Point which represent its coordinate on the map. The key is the String and the coordinate is the value of the map. The map only saves the address book of the sensors that required to be read. So we take List<Sensor> as input parameter. Also the translator is required. The App class will initialise a JsonServer instance and parse it to the method which is helped to do the translation. The code is straight forward. We just go through the Sensor list by a for-loop.

- The `OutputPath ( List<Route> flightpath, String day, String month, String year, Map<String,Point> W3Wmap, List<Sensors> task )` method. It is the very method that creates the readings-YYYY-MM-DD.geojson file and flightpath-YYYY-MM-DD.txt file. YYYY is the `String year`, MM is `String month`, DD is `String day`. Flight path in the type of `List<Route>` is to provide the "touched" Sensor information and the route of the drone to "touch" those Sensors. We fetch the start point, end point information to draw a lineString which is the path. If a Route "touches" a Sensor which is required to be read, we will use `colourConvert()` method to handle the Sensor

information and parse the info to a Point geometry. There is also a task parameter which is `List<Sensor>`. We will simply compare the number of Sensors the drone "touched" and the number of Sensors in the task. If there is any Sensor that not be visited, we will check which Sensor is the unvisited one by a for-loop and also save it as a Point geometry with a grey marker and no symbol. The `W3Wmap` of type `Map<String,Point>` helps to translate the W3W into coordinates which allows the Point being visualised on the geoJson map. The geoJson file is ready once we have added these features into a featureCollection and call the above `createGeoJsonFile()` method.

We also fetch the start point, angle, end point, visited Sensor information of each Route(step) and concat them into a String with the number of step at the front. If there is no sensor visited in this step, the sensor part should be a "null". If it does, it should be the "word1.word2.word3" String which is the Sensor.location to represent the sensor. Summarising every step sequentially then we can carry out the flight path text file. There is no separate method for writing these text files as for those geojson files. So I initialised a File and a writer to create the txt file.

- The `main( String [ ] args )` method. As its name, it is the centre controller function of the Application. The input parameter should be a String array of length 7: the `day` in DD, the `month` in MM, the `year` in YYYY, the `longitude` of the starting point, the `latitude` of the starting point, the `seed` which is a random number that might be used by the algorithm and the http localhost `port` number. The first three String composite a specific date which will be put into the JsonServer getSensor() method to read out the list of sensors required to be read today. Hence a JsonServer should be initialised in this main method, the port String is required for JsonServer instance initialisation. We also call the method in JsonServer for the NoFlyZone information. Then we should look at the Drone. The initialisation of a zone required the NoFlyZone information, the list of sensors and also the starting point of the drone that the drone can know from where it starts its path. The seed element is not being use in my application.

#2 Sensor.java

It has 3 private attributes:

- The `location` of the Sensor which is a String. The String should be in a what3words' format.

- The `reading` of the Sensor which is also String type. The reading should be in range 0 to 256 if we read it as a double or "NaN" or "null". The worse the air quality the higher the reading. If it is "NaN" or "null", the sensor might run out of the battery and powered off:

- Sensor's `battery` has maximum 100.00 which is fully charged and minimum 0.00 which means the Sensor runs out of power. It is a double. The rule is if the sensor's battery in lower than 10 then the air-quality-reading is not trust worthy.

For methods in Sensor, it only has the getters and setters for all attributes.

#3 NoFlyZone.java

A NoFlyZone instance representing a building which is too high that the drone can not fly over it. The class has 2 private attributes:

- A list of list of Point which is called `coordinates` . The inner list is a list of the zone's vertices in sequential order. Note that the first vertex will be mentioned again at the last. So a triangle-shaped NoFlyZone which has 3 vertices {A, B, C} does has the coordinates {{A, B, C, A}}. The attribute is set to final because the buildings floor space is fixes. Extension projects are not under consideration.

- The `name` of the building. It used to be set to final before I write down this paragraph. However, since I suddenly recall that David Hume Tower has changed its name to 40 George Square……

The NoFlyZone class has two methods except for the getters:

- The `getBoundaries()` method will return a list of lineString. That is the list of edges of the zones. The edges are carried out by simply going through a for loop that pairs two vertices in a sequential order.

- The `getRegion()` method will return a Polygon which is the area circled by its edges. The method is not in used now but used for testing— it helped to visualise the barriers on geojson.io and facilitate me to check whether my drone fly over this NoFlyZone — which is not acceptable.

#4 W3W.java

Each W3W class represent a What3Words object. It has 6 private attributes in total:

- A String which is the `country` name in where this location is.

- The `square` area which is of type Square that can be represent by this W3W.

- `word` ("Word1.Word2.Word3") which is the name of this W3W object .

- The single point but of type MyPoint — the exact `coordinates` of the Sensor location.

- `nearestPlace` which is the city it locates at.

- What language is used by the naming of W3W.

- The official website showing this W3W's address which the attribute's name is `map` .

For methods, It only contains the getters and setters. However, there are 2 classes defined in the W3W class:

#5 MyPoint.java

I create this class in order to facilitate the Json String to Class object parsing process. The class has two private attributes:

- `lng` which is a double representing the longitude of the address point.

- `lat` which is a double representing the latitude of the address point.

It also has a public method:

- `getP()` which is a Point-generating method which takes lng and lat as coordinates and returns a Point. lng and lat are private so we can't access them directly but we can get them both through this public method.

#6 Square.java

A Square object represent a real square area on the map. The square is confined by its 2 private attributes:

- southwest of type MyPoint which is the southwest corner of the square.

- northeast of type MyPoint which is the northeast corner of the square.

Square do not has any other method except for the getters. They do not have setters because each W3W object will be initialised by the JsonServer which all the attributes are filled and fixed. A Square object exits when the W3W object is generated by the JsonServer.

#7 JsonServer.java

This is the class which handles all WebServer operations. The class has 2 private attributes:

- A HttpClient type called `client` which is parsed in the constructor. It is set to final because one JsonServer only has one client and can not be replaced.

- A String-format integer which is the `port` number. It is also set to final. The port number can not be changed, or it will throw java.net.ConnectException.

Then there are 3 methods for 3 web content reading queries. They are all public since they are expected to be called by the App not used by itself. All of them are able to throw Interrupted-Exception and IOException.

- `getNoFlyZone()` method returns a list of NoFlyZone element. The noflyzones.geojson file in *buildings* folder will be read. The method sends request to the webServer and get response from it. The geoJson file will be read and the Json String which is the content in the file will be translated and the informations will be handed out to a list of NoFlyZone instances as their attributes. The function is able to catch java.net.ConnectException which is caused by the port

number mismatching problem. If “Invalid Port!!” Is shown on the screen, that means you need to check your port number.

- The `getW3W ( String words )` method which is for W3W element translating. The String is in a “word1.word2.word3” formate and the JsonServer will composite a pathname which is the combination of these three words navigating to *words* folder to find the Json file which includes its location in longitude, location in latitude, name, and other related details. A W3W instance will be generated.These method is operated under the same principle as above one. It can handle the port-number-mismatching problem as well.

- `getSensors ( String days, String month, String year )` is operated under the same principle. The return type is a list of Sensors. Days + month + year provide a specific date and navigate the JsonServer to find the sensor readings on that day in the *maps* folder. What’s more, except for catching the java.net.ConnectionException, if the date does not exist, for example you are looking for the air quality readings in Edinburgh on the 30th of February, 1988, the JsonServer is able to catch java.lang.IllegalStateException and tells you it is an “Invalid Date!!”.

#8 Drone.java

It is the internal control class of the drone. The “find path” algorithm is embedded in this class. This class has 6 private attributes:

- The private final `zones` of type List<NoFlyZone>. It is one of the input parameter which is needed for initialising a Drone instance. It is of final state because it will not change since it is defined. The no-fly-area can not once the task is assigned to the drone and the drone is started.

- The private final `goal` of type List<Sensor>. As above, it is one of the input parameter which is given as the Drone instance being generated. It is also final since the task should not be changed once the drone is start either.

- The private final `startingpoint` of type Point. As above, it is also parsed by Drone’s constructor. Just like the naming, startingpoint is the starting point of the drone. It is final because the drone could not teleport from one point to another :)

- The private final `W3Wmap` Map<String, Point> of type Point. It is parsed by Drone’s constructor as well. It provides information about the coordinates of Sensors which facilitate the distance calculating and avoid calling the WebServer multiple times which may leads to java.connection.exception.

- The private final `step` of type double. It decides the length of each step. It is final because length of one step is fixed which is 0.0003 degree. The drone can only go a full step per move.

- The private `Battery` of type int. The Battery is the 150 maximum and the battery should minus 1 per step moved. This means the drone should try to visit all the sensors in 150 steps. Once the drone is out of battery, the drone can not move any more and should be charged if it want to try again :(

What’s more, there are also a constructor and 11 methods in the Class:

▶ 2 of them are the battery monitoring method which are private because the battery can only be monitored by the drone itself:

- The **`hasBattery()`** method which will return true if it has battery > 0 else return false.

- The **`charging()`** method which return nothing but will charge the drone to full battery which is 150 once you call it.

▶ 2 of them are the path finding method, they are generated due to the design of Algorithm:

- The **`tourValue( List<Sensor> )`** method. It provides all the locations that drone need to visited in this task. The method use a for loop to calculate the cost between 2 Sensors in the list in sequential order. The cost between fly between each pair of Sensors is calculated by the following methods:

- The **`cost( Point A, Point B )`** method. It will return a list of Route which is required to pass through from the location of A to the location of B. At the same time, this method will call hasBattery( ) method to check drone's power state and reduce the drone battery each move. There is also a if-condition function in the method — if there is no barrier between 2 points, the drone will transfer the path-finding task to a no-barrier-path-finding-method:

- The **`flyStraight( Point A, Point B )`** method. The method handles the cases that there is no barrier on the way from A to B in the initial state. It has also been assigned functionality to avoid barriers if the drone crashes onto any NoFlyZone in some simple cases. The algorithm will be talked about later in the Algorithm part. It also calls the hasBattery() method to check battery and reduce the battery by 1 each time it moves.

▶ 4 of them are the pruning method which all return boolean:

- The **`noBarrier( Point A, Point B )`** method. It returns false if the line segment connecting A and B intersects with any of the edge of the NoFlyZones.

- The **`pnpoly()`** method. If the point is in any NoFlyZone polygon, the function will return true, else return false.

- The **`inConfArea()`** method. If the drone flies out of the confinement area, the function will return false. The confinement area is a rectangle that all the sensors are in the area and drones can not fly out of the area.

- The **`isVisited( List<Point>, Point A )`** method. It is the pruning method for the cost ( ) method. If A is visited before then the point is not being considered to visit again. The visited points are given by the parameter of type List<Point>. The method return true if A is visited.

▶ 3 of them are the optimisation purposed method:

- The **`roughPath( List<Sensor> )`** method. It will rearrange the permutation of the sensors which may reduce the number of steps of the flight path in total. The permutation of the

sensor is rearranged by Greedy approach.The details will be talked about later in the Algorithm Description section.

- The **`flyPath ( List<Sensor> )`** method, the trigger of all other functions in the drone. It will call the roughPath ( ) method first for a more advisable sensor order. Then it will start a twoOpt Algorithm in order to get a shorter flight path. It will call the above tourValue( ) method. It returns List<Route> and the value will be parsed back to APP class for file generated use.

- The **`reverseHeuristic ( List<Sensor> , int i, int j )`** method. It is the helper function of the twoOpt Heuristic method. The parameter i and j is the marker in the sensor list. The order of the Sensor will be reversed. The method compared the length of the flight path of the original one and reversed one by applying tourValue ( ), returning the shorter list of sensor.

#9 Route.java

A route instance carries the information about one step in the path. It has 4 private attributes:

- The start point of this step **`Start`** which is a Point.

- The end point of the step **`End`** which is a Point.

- The direction it flies from the Start point to the End point **`Angle`** which is an integer. The angle should be a number which is the multiple of 10. { Angle = x * 10 | 0 <= x < 36, x ∈ N }.

- The **`endS`** which is a Sensor. If the drone stop at the end point End and take the readings of a specific Sensor, the Sensor will be parsed as the endS.

Since all the attributes are private, I have generate the getters and the setters for each attribute.Route class has only the getter and setter methods.

#10 ConfinementArea.java

This class only has 4 private final attributes which means the already defined itself. This means every ConfinementArea is representing the same area even you initialised a lot.

- the **`west`** edge of the area which is a double.

- the **`east`** edge of the area which is a double.

- the **`north`** edge of the area which is a double.

- the **`south`** edge of the area which is a double.

Drone Control Algorithm

Requirements

1. Fly step by step which a step is 0.0003 degree;
2. The flying angle should obey { angle = x * 10 | 0 <= x < 36, x ∈ N }
3. Read the sensors within 0.0002 degree;
4. Fly from the starting point;
5. Return to a address within 0.0003 degree to the starting point;
6. Finish readings-taking in 150 steps;
7. Visit sensors as many as possible;
8. No out of Confinement Area/ fly over NoFlyZone

Preconditions

Since the drone is power off once it runs out of battery, after flying 150 steps the drone will be powered off, I assume the drone has 150 level battery. In my implementation, the battery will reduced by 1 each move. The drone will check the battery level by *hasBattery( )* method every time after taking a step and get the drone charged by *charging( )* method if we want to start the drone again. [Requirement 6]

From Point to Point

Firstly, assume that the drone only needs to fly from Point A to Point B.

The *cost( )* function provides the function to find path from A to B. My idea is using **Breadth First Search** (BFS) to find the path. Each path should be consist of one or more steps. I first create a BFS tree and expand the tree in breadth first order. A node (which is the **start point** of this step) can expand to 36 leafs (the **end point** of this step) which depends on the step length and the direction of each step. Each step is set to 0.0003 degrees [Requirement 1]. After calculating the straight-line **direction** from start to the end and rounded it into a **multiple of 10** in degree [Requirement 2], we start to adjust the flying angle by add on **penalty**. The penalty also should be the multiple of 10 in degree [Requirement 2]. I have create a list of acceptable penalty of size 36: [0,10,-10,20,-20…170,-170,180] which represent **36 distinct directions**. The angles are oscillating centred at a given direction in order to **aim at the target** and decreasing the time complexity. The algorithm will go through a for loop for the node expansion and a while loop is applying for continuously doing the expansion. Firstly, the start point is defined by coping the value of Point A out of the while loop. At the End of the while loop, the branch we have expanded just now will **be removed**, current position will be added into **visited list** and the **start** point will **be replaced** by the expanded end points. The further expansion will follow on in the next while-loop.

On this basis, there are several **pruning** conditions must be applied to the node expanding since those end points are not acceptable: check if the end point is **out of the confinement area** by *inConfArea( )*; check if the end point is **in the NoFlyZones** by *pnpoly( )*; the end point may not in the NoFlyZones but still need to check if the segment connecting start point and end point of this step has **intersected** with any of the NoFlyZones' edges by *noBarrier( )* [Requirement 8].

What else, I have also applied a **pruning** condition to the BFS search which can reduced the branches massively without breaking the functionality. Since the drone has 36 options for the next

point and there are definitely some points within the range of the previous points, I cut off the branches which their leafs are **visited** points. The point is visited or not is check by the method *isVisited( )*. The points which are within 0.0010 degree distance with any of the expanded points are regarded as visited.

Then the BFS needs a if condition to **break** the while loop and return the List<Routes> as result.

However, I do not simply use BFS to find the right way. Applying BFS is **time consuming and redundant** if there is no barriers on the way. If there is no NoFlyZones on the lineString connecting the start and B, we will summarise current branch from root to leaf to a Route list. Need to be aware that there is **loss of significance** when doing *toRadians( )* and *toDegrees( )* repeatedly when recalling the direction of each step, we need to round the decimal to 10x integer here. Go on summarisation in a for-loop and call *hasBattery( )* to **check the Battery** every time. If the drone still has power, the Route will be recorded by the drone and battery level minus 1. If the drone is out of battery, the method will return a **null** [Requirement 6]. If the drone is still powered on after finished previous steps, it jumps to another method which is provided for **no-barrier-path-finding** — *flyStraight( )*. The start point is parsed into *flyStraight( )* as the formal parameter Point A, and the another formal parameter Point B is still the goal point Point B in the *cost( )* function.

In *flyStraight( )* there is also a while loop driving the drone going step by step. First initialise the **start** point as Point A. We calculate the differences in longitude and latitude between the A and B and round the **angle** to an integer which is a multiple of 10 by *Math.Round( )* [Requirement 2]. The drone flies along this direction. Each step is still 0.0003 degree. After carrying out the end point of this step, we need to calculate the **distance** between drone's current position and the goal B. By replacing the **start** point by the **current position**, the while loop drives the drone going for the next step. Before the stepping action is token, the drone **battery** still need to be checked and reduced by one. If out of battery, method will return **null** [Requirement 6].

The **break** condition of this method is to check whether the distance between drone's current position and B's is within 0.0002 degrees [Requirement 3].

However, we can not ignore the cases that the drone crashes on the NoFlyZone or the edge of the confinement area because it is not going the straight line from A to B as we imagine. There is small **deviation** to the real angle and theoretical angle because the drone can only fly along the direction of multiple of 10 in degree. Hence we need a **deviation compensator**: I named it ***roundfactor***. The principle is: add 10° to the angle if the previous flying angle is **rounds down** by *Math.round( )* method applying to the theoretical angle or minus 10° if the previous angle **rounds up**. This can guarantee the drone won't **successively offtrack**. In general, the BFS is able to prevent the drone flying into blind ended alley and sends the drone to the wider road. If the cost BFS approach and deviation compensation still failed to prevent the drone from crashing, there is still an **angle-penalised-for-loop** prepared for those poor losing drones. As the design in BFS, the drone has 36 options and should choose the least penalised one. Every time the drone should be test by those three boolean method to check if the end point is valid [Requirement 8].

From Sensor to Sensor

The find path process from Point to Point is finished, now we look at how to move from sensors to sensors and the whole sensors list.

The difference is now we need to get the flight path of **successive goal point B** with **changing start point A**. They are implemented in the *tourValue( )* method. As the drone wants to fly from the **starting point** to the **first sensor** [Requirement 4], we parse the starting point as the parameter A and position of first sensor as B. It will return List<Route> which contains previous path info. Note that in above methods, we haven't take sensors readings. The drone just stops near the first sensor. So we need to read the sensor by adding this sensor to the **EndS** attributes of the **last Route in list**. Then as the drone's current parking position is saved in the **End** attributes of the **last Route in list**, we can start a new trip by considering the **current parking position** as the **start** and second Sensor's location as goal and applying *cost( )* to them again. Repeat it in the same way in a for loop which goes from the first Sensor to the last and finally to the starting point [Requirement 5], we can get the full flight path!

There is a *charging( )* method for the drone. Once you want to start the drone by calling *tourValue( )* method, the *charging( )* method ensures that the battery is **full** for this flight.

Read All Sensors in 150 Steps

As my baby drone knows how to visited all the points, the question is how to ensure that we can get the full readings and the drone will not shut down on the way.

My idea is: delete a sensor in the goal list if the drone can not visit all, and delete another one again if the drone still can not visit them all. Every time the **first sensor** in the given list dies a heroic death:) I wrote a **recursion** in the the *flyPath( )* method. If the returned result is null since the drone shut down on the way, it will call itself with a pruning Sensor list as the parameter and run the whole process again. The drone will fly back once it found that the work is not overload anymore [Requirement 6].

Minimus the Number of Steps

However, it is not enough for my clever drone. It wants to go-get the full readings. Hence 2 algorithms are applied to optimise the path planning. The first one is **Two-Opt Heuristic** which is embedded in the *flyPath( )* method. The helper function *reverseHeuristic( )* **reverses** the order of certain sublist of the sensors multiple times and refreshes the optimised sensor order frequently by returning the sensor list which has **shortest path** each time. The second one it a **Greedy** approach which the method is call *roughPath( )* — roughly sort the sensors for a shorter flight path. The method sorts the sensor list at the very beginning. It loops the Sensors in list one by one and finds the **closest unvisited Sensors**. Closeness is measured as the **straight-line distance** ignoring NoFlyZones. It will dramatically **reduce the time** costed by Two-Opt Heuristic and **optimise the flight path** result since Two-Opt can be more targeted as it is handling a pre-manipulated target [Requirement 7].

> "This is all of my algorithms. There are still several ways to optimise the work. I think functionality of the flyStraight( ) method could be more efficient and advisable if I use Depth-First-Search instead. I am trying to do so but it is really tricky for me to use DFS since I am not very good at recursion. The ConfinementArea class and Square class have a lot in common. Maybe the structure looks more well organised if I use an Interface to represent the relationships between."
>
> "However, my drone has done super great job already!"